高等学校计算机专业规划教材

Python程序设计教程



邱仲潘　刘燕文　王水德　编著

清华大学出版社

高等学校计算机专业规划教材

Python 程序设计教程

邱仲潘　刘燕文　王水德　编著

清华大学出版社
北　京

内 容 简 介

本书层次鲜明、结构严谨、内容翔实，由浅入深介绍 Python 程序设计的方方面面。最后一章将前面讲述的内容应用到项目中，并以模板的形式介绍项目的开发过程，理论联系实际项目，既适合初学者夯实基础，又能帮助 Python 程序员提升技能。

本书适合各类大中专学校学生作为教材，也可以作为程序员自学读物。

图书在版编目(CIP)数据

Python 程序设计教程/邱仲潘，刘燕文，王水德编著. —北京：清华大学出版社，2016
(高等学校计算机专业规划教材)
ISBN 978-7-302-45098-6

Ⅰ. ①P…　Ⅱ. ①邱…　②刘…　③王　Ⅲ. ①软件工具－程序设计－教材　Ⅳ. ①TP311.56

中国版本图书馆 CIP 数据核字(2016)第 227117 号

责任编辑：龙启铭　薛　阳
封面设计：何凤霞
责任校对：徐俊伟
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn,010-62795954
印 装 者：北京鑫海金澳胶印有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：19.5　　**字　　数**：462 千字
版　　次：2016 年 11 月第 1 版　　**印　　次**：2016 年 11 月第1 次印刷
印　　数：1～2000
定　　价：39.50 元

产品编号：066022-01



前言

Python已经成为最受欢迎的程序设计语言之一。2011年1月，它被TIOBE编程语言排行榜评为2010年度语言。自从2004年以后，Python的使用率呈线性增长。

由于Python语言的简洁、易读以及可扩展性，在国外用Python做科学计算的研究机构日益增多，一些知名大学已经采用Python教授程序设计课程。例如卡耐基梅隆大学的编程基础和麻省理工学院的计算机科学及编程导论就使用Python语言讲授。众多开源的科学计算软件包都提供了Python的调用接口，例如著名的计算机视觉库OpenCV、三维可视化库VTK、医学图像处理库ITK。而Python专用的科学计算扩展库就更多了，例如NumPy、SciPy和Matplotlib，它们分别为Python提供了数值计算、科学计算以及绘图功能。因此Python语言及其众多的扩展库所构成的开发环境十分适合工程技术、科研人员处理实验数据、制作图表，甚至开发科学计算应用程序。

本书最后一章将前面讲述的内容应用到项目中，并以模板的形式介绍项目的开发过程，既适合初学者夯实基础，又能帮助Python程序员提升技能。

编　者

2016年5月


目 录

第3章 程序流程控制 /41

第4章 序列：字符串、列表和元组 /67

第 5 章 映射和集合类型 /96

第 6 章 函数 /120

第 9 章 异常 /210

第 10 章 文件 /240

第11章 项目开发实例 /262

第1章 Python语言简介

本章学习目标

- 了解 Python 语言的发展历史
- 掌握 Python 语言的特点
- 了解 Python 的应用
- 熟练掌握 Python 的安装
- 掌握第一个 Python 程序

本章先向读者介绍一些有关 Python 的背景知识，什么是 Python 以及它的发展历史，然后介绍 Python 语言的特色、应用领域，在读者对 Python 语言有一定的了解之后，紧接着介绍 Python 的安装以及第一个 Python 程序，最后，本章末尾给出的练习题将使读者进一步巩固本章重要的知识点。

1.1 什么是 Python 语言

Python 是一种简单易学、面向对象、解释型的计算机程序设计语言，它既具备传统编译型程序设计语言的强大功能，又在某种程度上具备比较简单的脚本和解析型程序设计语言的易用性。其丰富的类库和简单易学的面向对象的编程特点深受初学者的喜爱，成为高等院校开设程序设计课程的主流编程语言之一，同时还因其具备可移植、可扩展等特性成为软件公司进行快速应用程序开发以及科研单位进行科学研究的主流编程语言。

Python 语言的语法简洁而清晰，具有丰富和强大的类库。它常被昵称为胶水语言，因为它能够很轻松地把用其他语言(尤其是 C/C++)编写的各种模块联结在一起。常见的一种应用情形是：使用 Python 快速生成程序的原型(有时甚至是程序的最终界面)，然后对其中有特别要求的部分，用更合适的语言改写，比如 3D 游戏中的图形渲染模块，速度要求非常高，就可以用 C++ 重写。Python 是一种高层次的结合了解释性、编译性、互动性和面向对象的脚本语言，具有很强的可读性。

现在，全世界差不多有六百多种编程语言，但流行的编程语言也就二十来种。比如 C、Java、.NET、PHP 等。我们不能说什么语言比较好，这几种编程语言各有千秋。C 语言是可以用来编写操作系统的贴近硬件的语言，所以，C 语言适合开发那些追求运行速度、充分发挥硬件性能的程序，而 Python 是用来编写应用程序的高级编程语言。

当用一种语言开始作真正的软件开发时，我们除了要编写代码外，还需要很多基本的已经写好的现成的东西，来帮助我们加快开发进度。比如说，要编写一个电子邮件客户端，如果先从最底层开始编写网络协议相关的代码，那估计一年半载也开发不出来。高级编程语言通常都会提供一个比较完善的基础代码库，让用户能直接调用，比如，针对电子邮件协议的 SMTP 库，针对桌面环境的 GUI 库，在这些已有的代码库的基础上开发，一个电子邮件客户端几天就能开发出来。

Python 为我们提供了非常完善的基础代码库，覆盖了网络、文件、GUI、数据库、文本等主要内容。用 Python 开发，许多功能不必从零开始编写，可以直接使用现成的模块。

除了内置的基础库外，Python 还有大量的第三方库，也就是别人开发好供我们直接调用的模块。当然，如果我们开发的代码封装得很好，让其他开发者很方便地调用，也可以作为第三方库提供给别人调用。

许多大型网站就是用 Python 开发的，例如国外著名的视频分享平台 YouTube 和 Instagram，还有国内的豆瓣，同时还包括 Google、Yahoo 等大型公司，甚至 NASA（National Aeronautics and Space Administration，美国国家航空航天局）都大量地使用 Python。

1.2　Python 语言的发展历史

Python 语言的开发工作由 Guido van Rossum 开始于 1989 年末，接下来转移至荷兰的 CWI(Centrum voor Wiskunde en Informatica，国家数学和计算机科学研究院)，并最终于 1991 年初公开发表。是什么促使他开发一门新的语言呢？一种程序设计语言的发明通常归结为两个原因：一是有一个资金充裕的大型研发项目作为支撑；二是因为缺乏某种软件工具而造成的困境，人们需要开发出一个新的工具来完成当时那些枯燥或者耗时的工作，而这些工作大部分又都是能够自动完成的。

Guido van Rossum 是 CWI 的一名研究员，他认识到高级教学语言 ABC(All Basic Code)因其语言不是开源的，不利于改进或扩展的重大缺点，因此，van Rossum 下定决心开发一种可扩展的高级编程语言，为其研究小组的 Amoeba 分布式操作系统执行管理任务。他从 ABC 汲取了大量的语法，并从系统编程语言 Modular-3 借鉴错误处理机制，开发出了一种能够通过类和编程接口进行扩展的高级编程语言，他将这种新语言命名为 Python(愿意为“大蟒蛇”)——来源于 BBC 当时正在热播的喜剧片 *Monty Python*。

自 1991 年初公开发行后，Python 开发者和用户社区逐渐壮大，Python 语言逐渐演变成一种成熟的、并获得良好支持的程序设计语言。Python 已经成为最受欢迎的程序设计语言之一。2011 年 1 月，Python 因在所有编程语言中占有最多市场份额，赢得 Tiobe 2010 年度语言大奖。自从 2004 年以后，Python 的使用率是呈线性增长的趋势。

由于 Python 语言的简洁、易读以及可扩展性，在国外用 Python 做科学计算的研究

机构日益增多，一些知名大学已经采用Python讲授程序设计课程。例如卡耐基梅隆大学的编程基础、麻省理工学院的计算机科学及编程导论就使用Python语言讲授。众多开源的科学计算软件包都提供了Python的调用接口，例如著名的计算机视觉库OpenCV、三维可视化库VTK、医学图像处理库ITK。而Python专用的科学计算扩展库就更多了，例如NumPy、SciPy和Matplotlib这三个十分经典的科学计算扩展库，它们分别为Python提供了数值计算、科学计算以及绘图功能。因此Python语言及其众多的扩展库所构成的开发环境十分适合工程技术、科研人员处理实验数据、制作图表，甚至开发科学计算应用程序。

1.3 Python语言的特点

一种语言之所以能够存在和发展，并具有较强的生命力，总是有其不同于(或优于)其他语言的特点。Python语言的主要特点如下。

1. 免费开源

像Java、PHP等语言都是开放源代码的，这些语言都得到了广大编程人员的认可，并对其进行改进，使其越来越完善。而Python也是考虑到长远的发展，采取了向公众开放源代码的策略，这样就能使任何一个Python语言的爱好者都能够自由发布这个软件的拷贝、阅读源代码并把它运用到新的开源软件中，这就是为什么Python语言如此优秀的原因之一——它一直被一些更加优秀的人不断改进。

2. 高级

程序设计语言每次更新换代都使我们进入更高一级。汇编语言可以直接对硬件进行操作，适合于对机器码很熟悉的开发人员。随后出现了FORTRAN、C和Passcal等语言，它们把计算任务带到一个更高的水平，并且开创了软件行业。这些语言又演化为如今的解析型系统设计语言C++和Java。再向上就是Tcl、Perl和Python等功能强大、能够进行系统调用的解析型脚本程序设计语言。这些语言都具有更高的数据结构，大大减少了项目中不可或缺的“程序框架”的开发时间。Python语言还建立了更为有效的数据类型，比如列表(list，即可变数组)和字典(hash table，即哈希表)等，减少开发时间的同时也减少了代码长度。

3. 易学

相对于其他编程语言，Python语言关键字少、结构简单、语法清晰，具有很强的伪代码特性，方便阅读，这样就使得程序设计初学者可以在更短的时间内轻松上手。

4. 易读

Python与其他语言显著的差异是：它没有其他语言通常用来定义变量、定义代码块和进行模式匹配的命令式符号。通常这些符号包括：美元符号($)、分号(;)等。没有这些符号，Python代码变得更加清晰和易于阅读。

5. 面向对象

像Java、C#语言一样，Python也支持面向对象编程，不同的是它还支持面向过程的

编程。面向对象的程序设计(Object Oriented Programming,OOP)为结构化和过程化程序设计语言增添了新的活力,面向对象编程技术的关键性观念是它将数据及对数据的操作行为组合在一起,作为一个相互依存、不可分割的整体——对象。而在面向过程的编程中,程序是由过程或可重用的函数模块来构建起来的。

6. 解释执行

Python 是一种解释型的语言,使用这种语言编写的程序,不需要编译成计算机可执行的二进制代码,而是直接从源代码运行程序。在计算机内部,像使用 C/C++ 等编译型语言编写的程序,必须通过编译器和不同的标记、选项把程序的源代码编译成计算机可执行的二进制语言。当运行程序时,连接/转载器软件再把程序从硬盘复制到内存中并且执行。而 Python 程序是通过 Python 解释器解释并执行的,Python 解释器把程序的源代码转换成称为字节码的中间形式,然后再把它翻译成计算机语言并执行,使得程序员无须关心程序如何编译、程序中用到的库如何加载等复杂问题。这样,使用 Python 将会更加简单,也更容易移植。

7. 灵活性

人们通常会把 Python 语言与批处理或 UNIX 系统下的 shell 脚本语言相提并论。简单的 shell 脚本可以用来处理简单的任务,shell 脚本的代码重用度很低,因此,它只能局限于小项目。而 Python 可以开发很大型的项目,用户可以不断地在各个项目中完善自己的代码,随时重用已写好的代码。Python 提倡简洁的代码设计、高级的数据结构和模块化的组件,这些特点可以让用户在扩大项目规模的同时,确保灵活性和一致性,并缩短必要的调试时间。

8. 可扩展性

Python 的可扩展性使得程序员能够灵活地附加程序,缩短开发周期,因为 Python 是基于 C 语言开发的,所以用 C/C++ 来编写 Python 的扩展功能。发展到现在,Python 也有基于 Java 实现的 Jython,从而使得 Python 可以在更多的语言中使用。

9. 嵌入性

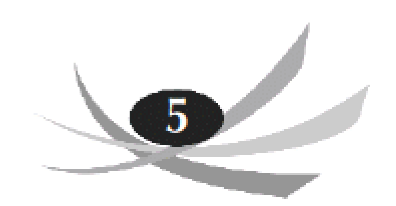
Python 的嵌入性是指它可以作为一种成熟的脚本语言,并且以一种很方便的方式嵌入到其他的程序中,比如 C/C++ 中。

10. 可移植性

Python 具有强大的可移植性,只需要把 Python 程序拷贝到另一台计算机上就可以很方便地移植到各种主流的系统平台中,这是因为 Python 是用 C 写的,又由于 C 的可移植性,使得 Python 可以运行在任何带有 ANSI C 编译器的平台上。尽管有一些针对不同平台开发的特有模块,但是在任何一个平台上用 Python 开发的通用软件都可以稍作修改或者原封不动地在其他平台上运行。这种可移植性既适用于不同的架构,也适用于不同的操作系统。

11. 易维护

源代码维护是软件开发生命周期的组成部分。Python 项目的成功很大程度上要归功于其源代码的易于维护,同时也因为 Python 本身就是易于学习和阅读的,从而使得用 Python 语言开发的项目具有易维护的特点。



12. 丰富的类库

Python是世界上具有标准库最大的编程语言。基于庞大的标准库，我们可以编写程序来处理各种工作，包括正则表达式、文档生成、单元测试、线程等功能。

13. 内存管理器

在程序开发过程中，我们会遇到像使用C/C++时要考虑的程序的内存管理问题。即使开发的是很小的程序，应用程序的修改和管理也需要程序员额外负责，这就需要开发者付出更多的精力。而在Python的程序开发过程中，Python解析器承担了程序的内存管理工作，使得程序员从内存事务处理中解脱出来，致力于程序功能的实现，从而减少错误，缩短开发周期。

1.4　Python语言的应用

由于Python语言具有简单易学、可扩展、可移植等优点，自2006年以来，Python已成为继C++、Java之后的第三种编程语言，更多地被应用到著名的搜索引擎，如Google，还有应用到曾经称霸智能手机市场的Nokia所采用的Symbian操作系统上，可见Python的应用领域非常广泛。表1-1介绍了Python语言的应用领域。

表1-1　Python语言的应用领域

应用领域	详细描述
系统编程	提供API编程接口，能够方便地进行系统维护和管理，是很多系统管理员理想的编程工具，是Linux系统下的标志性语言之一
图形处理	含有庞大的对诸如PIL、Tkinter等图形类库的支持，能够方便地进行图形处理
数字处理	NumPy扩展提供了大量与许多标准数学库对应的接口，可以方便地处理数学问题
文本处理	Python提供了很多模块，如re模块能够处理正则表达式，又如SGML、XML分析模块可进行文本的编程开发
数据库编程	通过Python DB-API(数据库应用程序编程接口)规范模块，可以与Microsoft SQL Server、Oracle、Sybase、DB2、MySQL、SQLite等数据库通信。Python自带的Gadfly模块可提供完整的SQL环境
网络编程	提供丰富的模块支持Socket编程，能够方便、快速地开发分布式应用程序
Web编程	支持HTML、XML等标记语言
多媒体应用	Python的PyGame模块可用于编写游戏软件，PyOpenGL模块则封装了OpenGL应用程序编程接口，能进行二维和三维图像处理

1.5　Python的安装

Python语言是跨平台的，它可以运行在Windows、MAC和各种Linux/UNIX系统上。在Windows上编写Python程序，可以轻松方便地移植到MAC和各种Linux/

UNIX 系统上。

在了解了 Python 的特点和应用领域之后，我们就可以进行 Python 语言开发的学习了。在学习之前，首先必须要知道如何获取 Python 开发工具、如何安装以及怎样启动 Python 开发工具，用 Python 编写的程序只有在安装了 Python 和配置好开发环境的前提下才能够运行，在这里我们将讲解如何获取、安装和启动 Python。

在告诉读者如何获取 Python 开发工具前，先给读者普及一些 Python 版本的知识。目前，Python 有两个版本系列，一个是 2. x 版，一个是 3. x 版，这两个版本是不兼容的，因为现在 Python 正在朝着 3. x 版本进化，在进化过程中，大量的针对 2. x 版本的代码要修改后才能运行，所以，目前有许多第三方库还暂时无法在 3. x 上使用。

为了保证用户的程序能够正常使用大量的第三方库，我们的教程仍以 2. x 版本为基础，确切地说，是目前官方网站中发布的最新版本 2. 7. 10。同时考虑到使用 Windows 系统的用户占了绝大部分，再者，初学者可能对 Linux 系统不熟悉，所以本节主要讲解基于 Windows 系统的安装。

1. Python 的获取

我们可以从 Python 的官方网站下载该软件。打开浏览器，在地址栏输入"https://www.python.org/"打开官方网站，选择 Downloads 菜单项，之后再单击 Python2. 7. 10（可能在读者看到本书时，Python 又已经出了新的版本，此时，可以单击 View the full list 查找到本书讲解的安装版本），如图 1-1 所示。

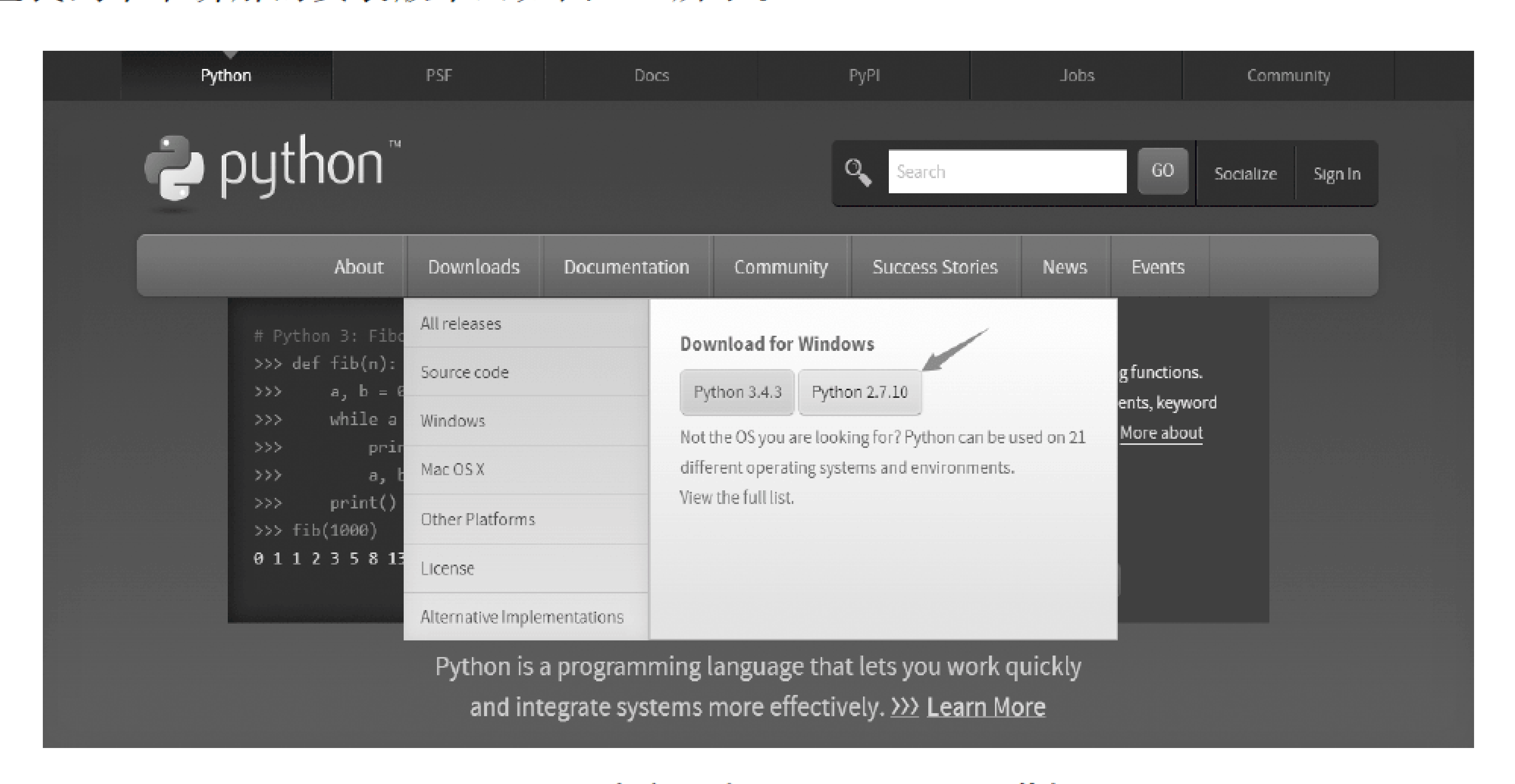


图 1-1 Python 官方网站 Python 2. 7. 10 下载提示图

2. Python 的安装

(1) 在下载目录中找到刚刚下载好的 Python 安装文件 python-2. 7. 10. msi，双击这个文件，会弹出 Python 安装程序的安装向导对话框，如图 1-2 所示。

(2) 在这里我们可以看到两个单选按钮，第一个 Install for all users 是为所有用户安装，第二个 Install just for me 是为个人用户安装。我们单击第一个单选按钮，然后单击 Next 按钮进入 Python 的安装路径设置界面，如图 1-3 所示。

(3) 选择安装路径。可以把路径更改为硬盘的任意路径(建议不要安装在有中文的

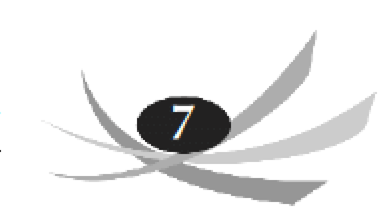




图 1-2　Python 安装程序向导对话框



图 1-3　Python 安装路径设置界面

路径下，否则可能会无法启动 Python）。在这我们把 Python 安装在 D:\Python27 目录下，选择好安装路径后，单击 Next 按钮，进入 Python 安装组件选择界面，如果不想自己手动配置环境变量，可以下拉右边的滚动条，单击 Add python.exe to Path 左边的小三角图标，选择 Will be installed on local hard drive，选择这一步，系统在安装 Python 时会自动把其环境变量配置好，我们建议要学会自己手动配置环境变量，如图 1-4 所示。

（4）单击 Next 按钮，进入 Python 工具包的安装界面，如图 1-5 所示。

（5）等程序安装完成后，会提示程序安装成功界面，最后单击 Finish 按钮完成 Python 的安装，如图 1-6 所示。



图 1-4　Python 安装组件选择界面



图 1-5　Python 工具包安装界面

3. Python 的环境配置

到这里我们已经成功安装好 Python 了。还记得安装 Python 的第三步吗，如果选择系统自动配置 Python 的环境变量，可以跳过 Python 的环境配置这一步，但这里还是很有必要讲一下如何手动配置环境变量，因为现在安装大多数语言的集成开发工具都需要配置环境变量。下面就说说如何配置 Python 的环境变量。

配置 Python 的环境变量有以下两种方式：

(1) 图形化操作的方式配置

在电脑的桌面上右击“计算机”图标(注意，本书教程中采用的是 Windows 8 系统，如



图 1-6　Python 安装完成界面

果用户采用的是 Windows 7 系统，可能会有小小的区别，但不会影响 Python 环境变量的配置），在弹出的快捷菜单中单击最下面的“属性”命令，会出现系统对话框，然后单击左边的“高级系统设置”，在弹出的“系统属性”对话框中单击“高级”选项卡，切换到系统高级设置界面，如图 1-7 所示。



图 1-7　系统属性高级设置界面

单击“环境变量”按钮，在弹出的“环境变量”对话框中会列出管理员用户变量和系统变量，如图 1-8 所示。



图 1-8　环境变量设置界面

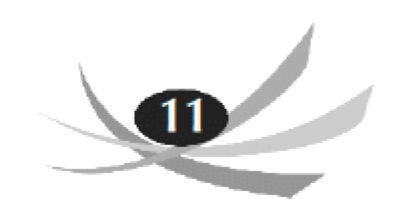
从“系统变量”列表框中找到并选中名为 Path 的变量后双击或单击下方的“编辑”按钮会出现“编辑系统变量”对话框，如图 1-9 所示。

图 1-9　编辑系统变量界面

在“变量值”文本框中最后面输入英文的分号以区分开前面已设置好的环境变量，再添加 Python 的安装路径 D:\Python27（根据自己的安装路径进行添加），单击“确定”按钮。通过以上几步就完成了 Python 环境变量的配置。

（2）命令行方式配置

通过 DOS 命令窗口配置 Python 环境变量比图形化操作方式要简单，只需一行命令即可，单击“开始”→“运行”选项，会弹出“运行”对话框，在该对话框中的“打开”文本框中输入“cmd”进入 DOS 命令窗口，输入“set path=%path%;D:\Python27”（注意：%符后



面的是英文的分号)回车即可,这行命令的作用是把 Python 的安装路径添加到当前的系统路径当中。

当正确安装了 Python,并配置了 Python 的环境变量后,我们就可以正常运行 Python 了。在这里我们可以通过两种方式来启动 Python:一种是使用命令行启动,另一种是使用 Python 的集成开发环境 IDLE。

4. Python 的启动方式

(1) Python 的命令行启动

单击"开始"→"运行"选项,会弹出"运行"对话框,在该对话框中的"打开"文本框中输入"Python",如图 1-10 所示。

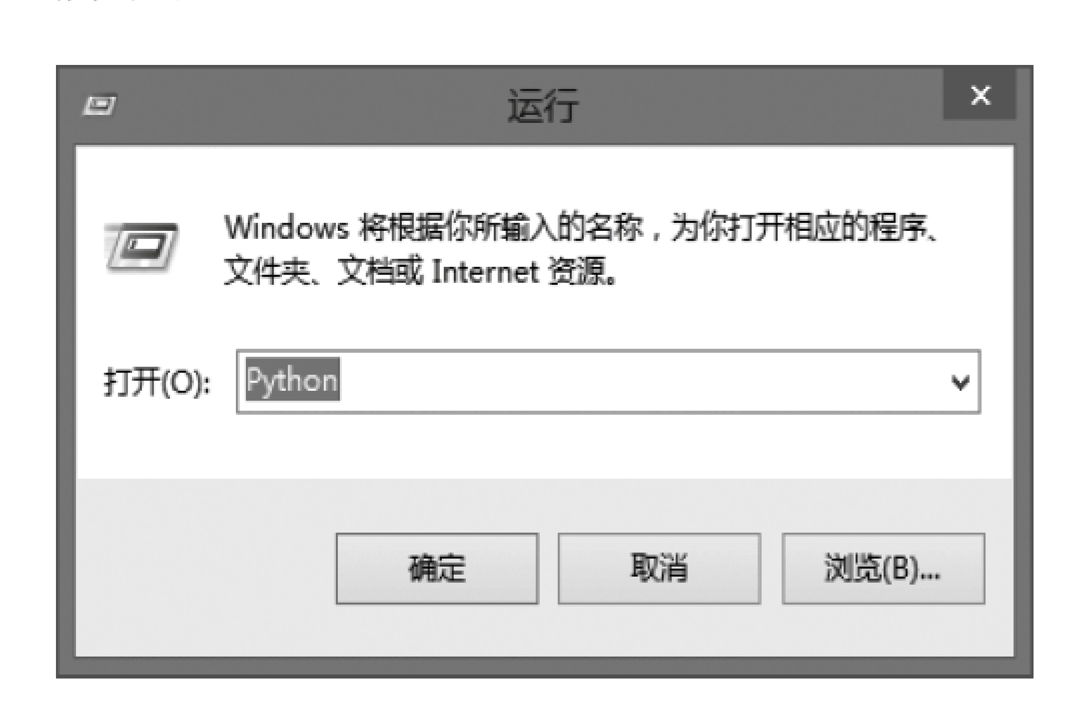


图 1-10 "运行"对话框

按回车键或单击"确定"按钮,如果出现如图 1-11 所示界面,则说明已成功在 DOS 系统下启动 Python 了。

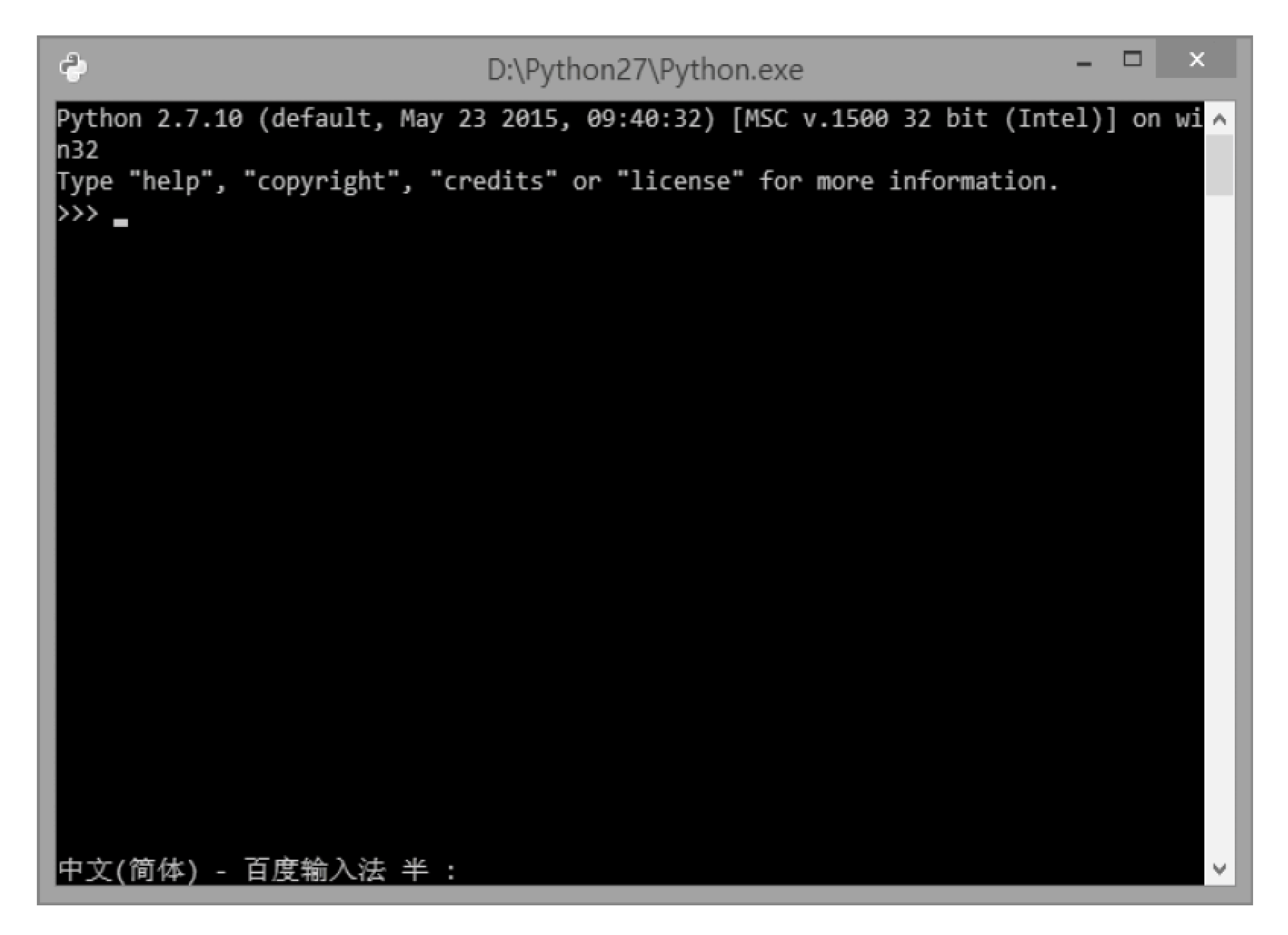


图 1-11 DOS 系统下启动 Python

如果出现图 1-12 所示界面,则说明环境变量配置错误(大部分是因为安装路径写错),需要重新配置。



图 1-12　启动 Python 出现的错误

(2) 使用 Python 集成开发环境启动

除了上面提到的使用 Python 命令行启动 Python 这种方式之外，我们也可以使用 Python 集成开发环境来启动 Python。单击“开始”菜单→“程序”→Python2.7→IDLE (Python GUI)启动 Python。如图 1-13 所示。

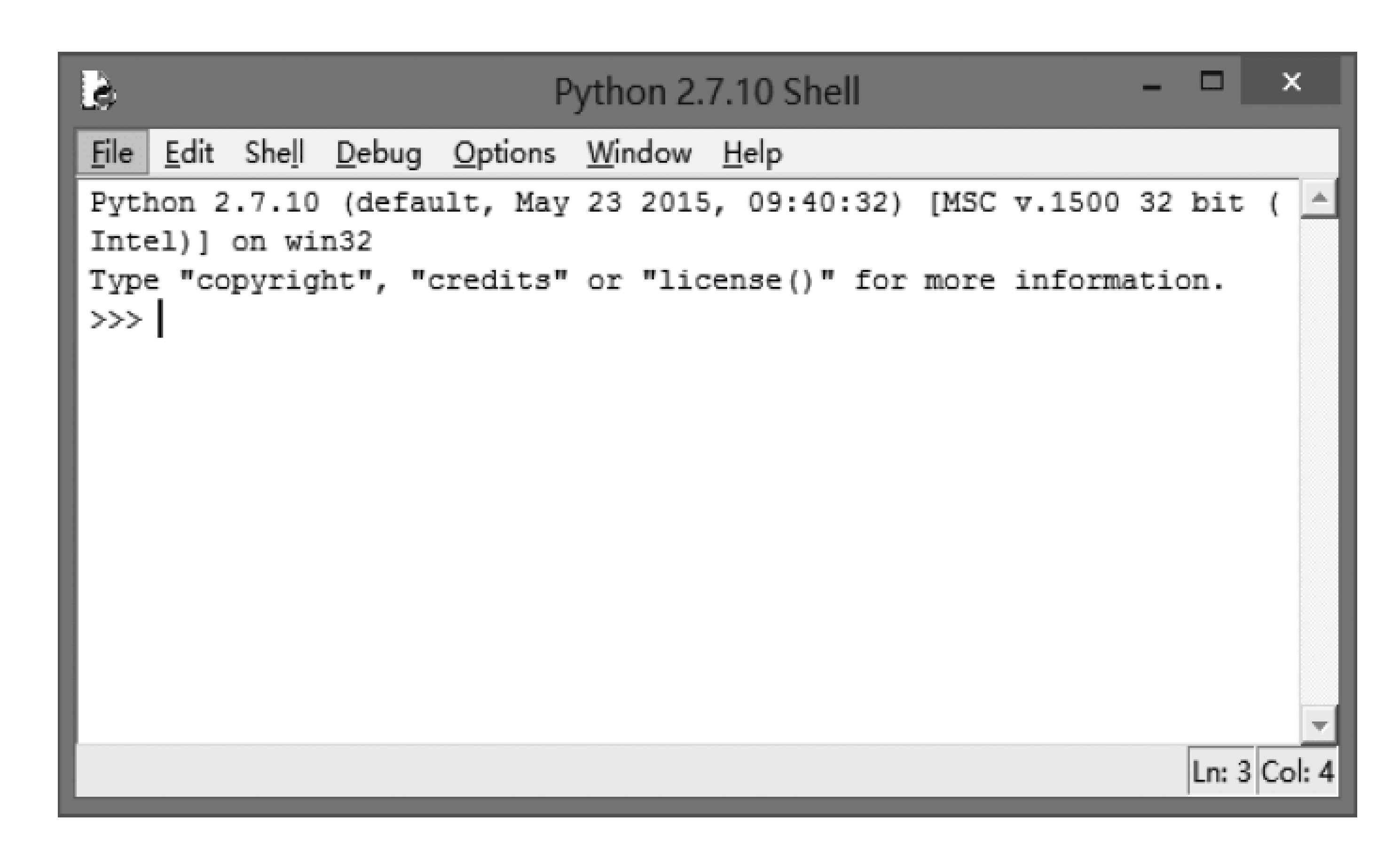


图 1-13　使用 IDLE 集成开发环境启动 Python

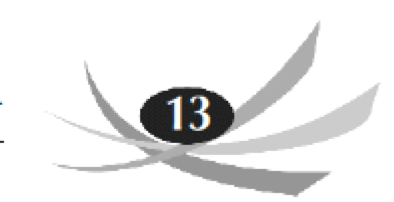
1.6　第一个 Python 程序

通过上一节的讲解，相信大家已经掌握了如何安装 Python 以及如何启动 Python 解析器，接下来我们就可以正式开始编写 Python 代码了。按照国际惯例，每学习一门语言，第一个程序都会讲解如何编写“hello world”。首先，我们通过 IDLE 集成开发环境启动 Python 解析器，然后在交互式环境的提示符“>>>”下输入 print “hello world”命令行后回车，即可看到 Python 解析器会输出 hello world。这行命令是调用了 Python 内置的一个 print 函数，在解析器上打印出引号里面的内容，如图 1-14 所示。

```
#例 1-1　编写 hello world
>>>print "hello world"
hello world
```

注意，主提示符“>>>”是提示我们要输入命令，所以，输入命令时不需要输入“>>>”。

在 Python 解析器中，我们编写的程序可以在交互式模式解析执行。在这种模式下，它主要根据主提示符来执行，Python 中的主提示符标记通常是三个大于号(>>>)，除此之外，还有从属提示符，以三个点来标记(...)。



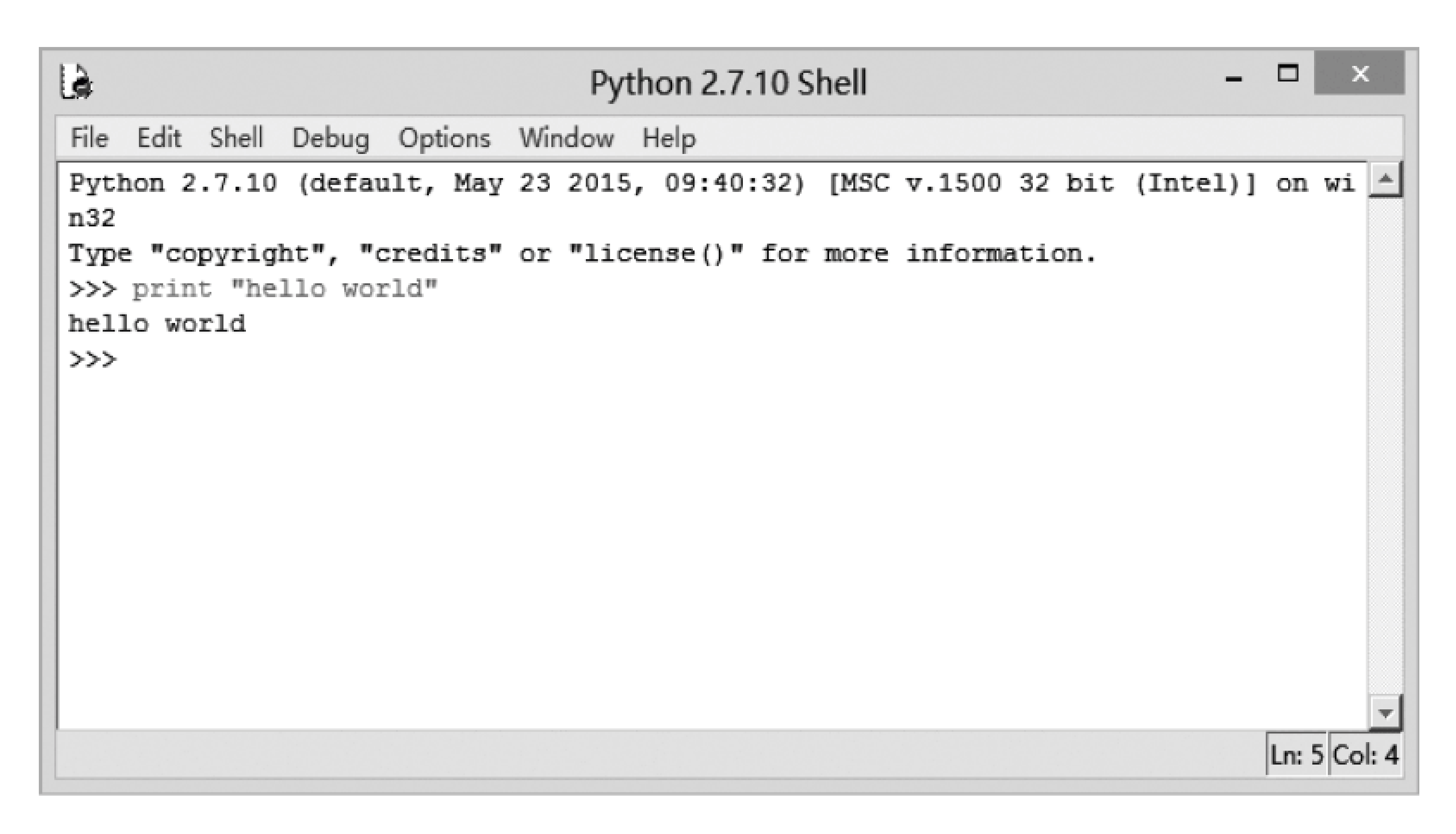


图 1-14　第一个 Python 程序

在交互式环境下输入的代码在退出 Python 解析器时会清空所有内容，所以，为了保存所编写的代码，提高代码的重用性，需要把代码编写在一个源文件中并保存起来，下次要想执行这些命令，只需在 DOS 命令窗口下输入该文件全名即可。接下来讲解如何创建源文件、编写输出 hello world 功能的代码并调用该文件执行输出 hello world 功能的代码。

(1) 在集成开发环境下启动的 Python Shell 窗口中单击 File→New File 或者通过快捷键 Ctrl＋N 创建一个 Python 的源文件。

(2) 在第(1)步创建的源文件中输入 print “hello world”。单击 File→Save 或者通过快捷键 Ctrl＋S 打开保存对话框，在文件名中输入任意的文件名，如 FirstPython，单击“保存”按钮。(Python 源文件名以.py 或.pyw 为后缀名)。

(3) 单击“开始”菜单→“运行”→输入“cmd”回车打开 DOS 命令窗口，然后切换到刚才保存 FirstPython 文件的所在目录，输入 FirstPython.py 回车即可看到在 DOS 系统输出 hello world，如图 1-15 所示。

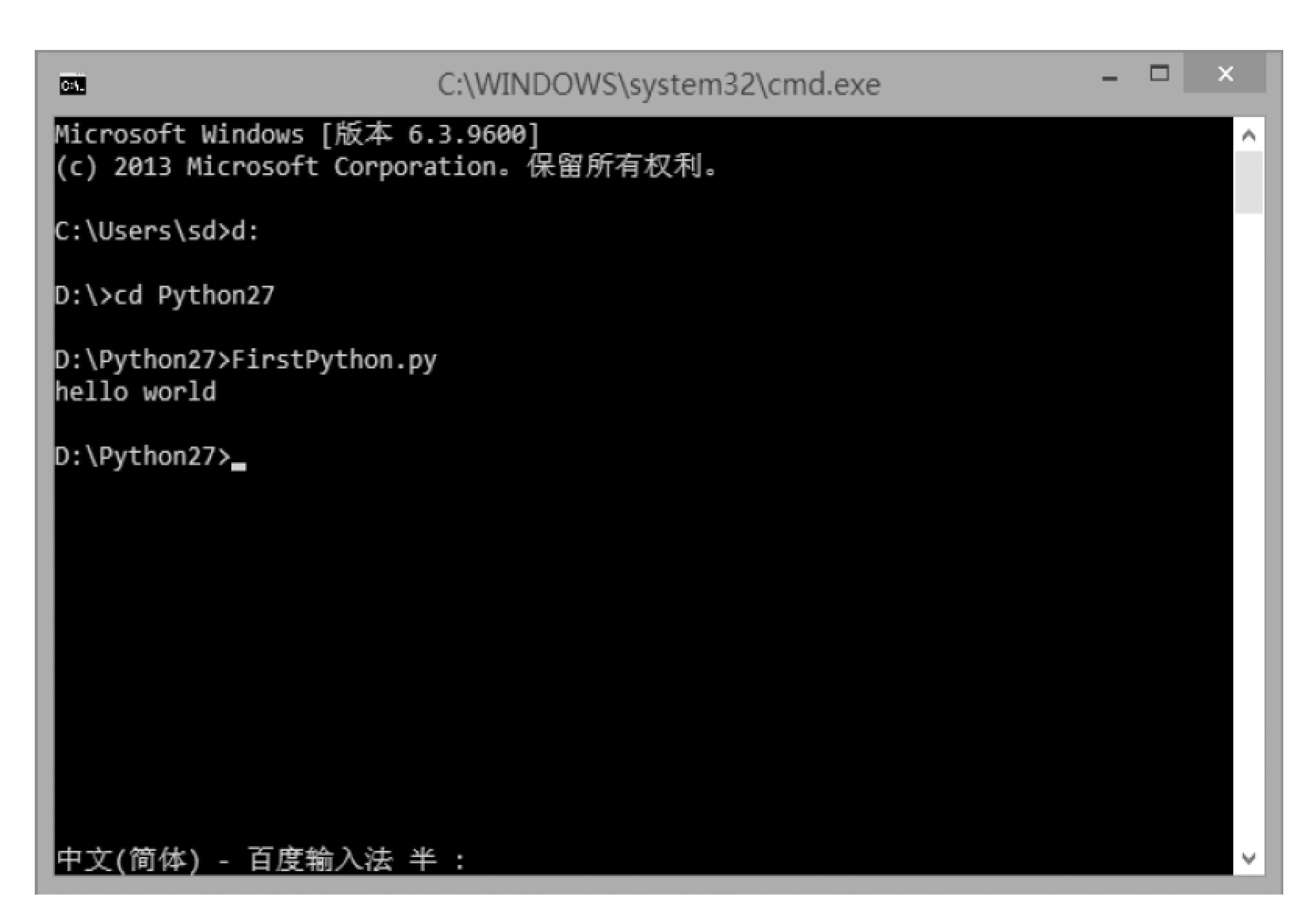


图 1-15　在 DOS 系统下执行已编写好的 Python 程序

1.7 本章小结

本章主要讲解了以下几个知识点：

(1) 什么是 Python。Python 是一种简单易学、面向对象、解释型的计算机程序设计语言，它提供了非常完善的基础代码库，大大加快了我们开发项目的速度，缩短了开发周期。

(2) Python 语言的发展历史。Python 语言由 Guido van Rossum 在参考了 ABC 语言，并从系统编程语言 Modular-3 借鉴错误处理机制于 1991 年初开发完成。由于其简单易学，功能强大，已成为了继 C++、Java 之后的第三大编程语言。

(3) Python 语言的特点。Python 语言具有免费开源、高级、易学易读、面向对象、解释执行、灵活性、可扩展性、嵌入性、可移植性、易维护、丰富的类库、内存管理器等特点。

(4) Python 语言的应用。Python 语言的应用非常广泛，包含有系统编程、图形处理、数字处理、文本处理、数据库编程、网络编程、Web 编程、多媒体应用等领域。

(5) Python 的安装。本节以 Windows 平台下的 Python 2.7.10 为例详细讲解了 Python 的安装过程，并配置环境变量以及启动 Python 的两种方式：一是 Python 的命令行启动；二是使用 Python 集成开发环境启动。

(6) 第一个 Python 程序。本节以输出 hello world 为例讲解了如何在交互式环境下编辑代码，同时还讲解了如何创建 Python 源文件，并在 DOS 命令行下执行已编好的 Python 源文件。

1.8 习 题

一、解答题

1. 什么是 Python？
2. Python 语言是谁开发的？他从哪种语言借鉴了错误处理机制？
3. Python 语言有什么特点？
4. Python 语言有哪些应用领域？

二、上机练习

1. 安装 Python 并配置环境变量。
2. 在交互式环境下用 print 函数输出“hello world”。
3. 创建一个 Python 的源文件，使用 print 函数输出你的基本信息，包含姓名、性别、年龄、住址等信息，并在 DOS 系统下执行该文件。

第 2 章 数据类型、运算符以及表达式

本章学习目标

- 了解三种最常用的标识符命名方法
- 理解 Python 中的输出格式
- 掌握 Python 的基本数据类型
- 掌握 Python 中的输入输出函数
- 掌握 Python 中的运算符和表达式

本章先介绍 Python 语言的简单数据类型，如整数、浮点型、布尔型。然后介绍 Python 语言的输入输出函数，即输入的 input 函数、raw_input 函数以及输出的 print 语句，紧接着介绍 Python 的运算符和表达式。在介绍相关知识点的过程中结合例子进行说明，以便让读者更好地理解，掌握知识点。最后，本章末尾给出的练习题将使读者进一步巩固本章重要的知识点。

2.1 数据类型

一个程序应包含算法和数据两个方面的内容。算法在后面章节会有详细讲解，数据是以某种特定的形式存在的(例如整型、浮点型、字符串等形式)。对数据的组织形式就称为数据结构，例如，数组就是一种数据结构。不同的计算机语言所允许定义和使用的数据结构是不同的。例如，C 语言提供了“结构体”这样一种数据结构，而 FORTRAN 语言就不提供这种数据结构。不同的数据结构，对处理某个问题时，所选的算法也会有所不同，最终的执行效率也会有所不同。

Python 语言的数据类型主要包括整型(int)、浮点型(float)、字符串(string)、布尔类型(bool)、列表(list)、元组(tuple)、集合(set)、字典(dictionary)。本章主要介绍整型、浮点型、布尔类型。字符串、列表、元组、集合和字典将分别放在第 4、5 章中介绍。在具体介绍这些数据类型之前，先讲解一下变量的一些基本知识。

2.1.1 变量

变量是一种使用方便的占位符，用于引用计算机内存地址。变量代表在内存中具有特定属性的一个存储单元，它用来存放数据，也就是变量的值，在程序运行期间，这些值是可以改变的。一个变量应该有一个名字，以便被引用。

和其他高级语言一样，在 Python 语言中用来对变量、函数、数组等数据对象命名的有效字符序列称为标识符(identifier)。

Python 语言规定标识符只能由字母、数字和下划线三种字符组成，且第一个字符必须为字母或下划线。下面列出的是合法的标识符：

amount，Sum，_rate，User_name，BASE，Li_Wang

下面给出的是不合法的标识符：

MR. White，$11，&name，1Varable

注意，Python 解析器将大写字母和小写字母认为是不同的字符。所以，对于 total 和 TOTAL，Python 解析器会解析成不同的两个变量名。一般情况下，为了符合人们日常阅读习惯，变量名用小写字母表示。

在定义变量和选择其他标识符时，应该做到“见名知意”，即选择有含义的单词或其缩写作为标识符，例如 year、month、day、amount、total 等，一般不要使用如 a，b，c 等含义不清晰的字符作为变量或其他标识符。此外，还需要注意，标识符不能和 Python 提供的关键字相同。下面列举出了 Python 2. x 版本和 Python 3. x 版的关键字。

Python 2. x 版本包含 31 个关键字：

```
and     as      assert  break   class   continue  def
del     elif    else    except  exec    finally   for
from    global  if      import  in      is        lambda
not     or      pass    print   raise   return    try
while   with    yield
```

Python 3. x 版本则有 33 个关键字：

```
False    None      True      and     as     assert   break
class    continue  def       del     elif   else     except
finally  for       from      global  if     import   in
is       lambda    nonlocal  not     or     pass     raise
return   try       while     with    yield
```

常用的关键字在后续章节中会陆续接触到。

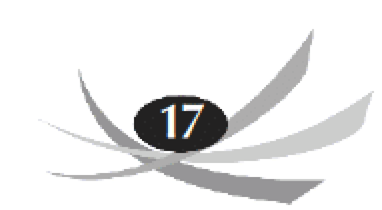
下面介绍三种最常用的命名方法。

(1) Pascal Notation(帕斯卡命名法)

这种命名法所有单词的第一个字母大写，其他字母小写。经常被用在定义类中。例如，StudentInformation 可以作为一个类名。

(2) Camel Notation(驼峰命名法)

这种命名法除了第一个单词外，所有单词第一个字母大写，其他字母小写。变量、函数等基本都采用这种命名方法，例如 userName、def judgeYear(year)。这种命名法正是由于其大小写错落有致，形状像驼峰而得名。驼峰命名法的另一种形式是使用下划线将两个单词隔开，单词均使用小写，例如 user_name、def judge_year(year)。



(3) Hungarian Notation(匈牙利命名法)

大部分流行语言,如Java、C#等都采用帕斯卡命名法和驼峰命名法。而对于传统的编程语言,如C、C++等,匈牙利命名法则用的较多,这种命名法因为由一名匈牙利程序员最初归纳而得名。

匈牙利命名法的基本规则是特性+描述。特性就是一个字母前缀,用于指定作用域、类型等信息,然后是变量的功能描述信息。功能描述信息是首字母大写的一个单词或多个单词的组合,该单词往往要指明变量的用途。

例如变量m_wndStatusBar,前缀m_代表类的成员,wnd也是前缀,代表变量对象的特性,这里wnd的意义是窗口,所有m_wnd表示窗口类的成员,而StatusBar则是变量的功能描述。在讲解完变量的基本知识后,接下来详细介绍Python的数据类型。

2.1.2　整型

1. 整型数据在内存中的存放形式

数据在内存中是以二进制形式存放的。如果定义了一个变量count,并把整数15赋给该变量:

```
>>>count=15     #把整数15赋给count变量
```

十进制数15的二进制形式为0000000000001111(假设一个整数占16位),Python 2.7为一个整型变量在内存中分配12个字节的存储单元。实际上,数值在内存中是以补码表示的。一个正整数的补码和该数的原码(即该数的二进制形式)相同。如果数值是负的,则该数的原码与补码会有所不同。求负数补码的方法是:将该数的绝对值的二进制形式按位取反再加1。例如求-15的补码的方法是:

(1) 取−15的绝对值15;

(2) 15的二进制形式为0000000000001111(假设一个整数占16位);

(3) 对0000000000001111取反得111111111110000;

(4) 再加1得111111111110001。

由此可得,在存放整型数的存储单元中,最左边的一位是表示符号的,该位为1,表示数值为负;该位为0,则表示数值为正。

关于补码的知识,感兴趣的读者可以查阅相关的资料。

注意到在count=15这条语句后面有#符,并跟着一段说明文字,这是Python语言的注释部分,注释可以用汉字或英文字符表示。注释只是给人看的,以增加程序的可读性,对程序的运行不起作用。一个好的程序应该要加上必要的注释。

2. 整型变量的分类

Python语言的整型相当于C语言中的long类型,在32位机器上,整型的位宽为32位,取值范围为−2147483648~2147483647;在64位系统上,整型的位宽通常为64位,取值范围为−9223372036854775808~9223372036854775807。

与C语言不同,Python的长整型没有指定位宽,也就是说,Python没有限制长整型数值的大小,实际上,由于机器内存有限,所以长整型数值不可能无限大。

在使用的过程中，我们如何区分长整型和整型数值呢？通常的做法是在数字尾部加上一个大写字母 L 或小写字母 l 以表示该整数是长整型的，例如：

```
>>>a=123456789L
>>>b=123456789l
```

因为小写字母 l 和数字 1 看上去很难区分，所以一般推荐使用大写字母 L。实际上，现在的 Python 解析器已经做了优化，如果整型数据发生溢出，Python 解析器会自动将整型数据转换为长整型数据，所以就算在长整型数据后面不加字母 L，对程序的运行结果也不会有什么影响，读者对它有一定的了解即可，以便在阅读他人写的程序时遇到数字后面带上 L 而不致感到茫然。

Python 中的整数不仅可以用十进制表示，也可以用八进制和十六进制表示。如果用八进制表示整数，数值前面要加上一个前缀“0”；如果用十六进制表示整数时，数字前面要加上前缀 0X 或 0x。例如，我们这里将整数 15 分别以八进制 017 和十六进制 0xf 的形式赋给整型变量 a 和 b，然后再以十进制的形式输出它们。

```
#例 2-1  八进制和十六进制数的使用
>>>a=0xf
>>>b=017
>>>print a
15
>>>print b
15
```

2.1.3 浮点型

1. 浮点数的表示方法

Python 语言中的浮点数（floating point number）就是平常所说的实数（real number）。

浮点数有两种表示形式。

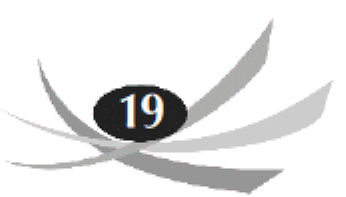
（1）十进制小数形式。它由数字和小数点组成。例如 3.14159、12.3、0.123 等。

（2）指数形式。如 1.23e2 或 1.23E2 都表示 1.23×10^2。但注意字母 e(或 E)之前必须有数字，且后面的指数必须为整数，例如 e1、1e2.3、.e 等都不是合法的指数形式。

Python 语言中的浮点型数据之所以称为浮点数，是因为按照指数(即科学记数法)表示时，一个浮点数的小数点位置是可变的，例如 1234 可以用 1.234×10^3、12.34×10^2、123.4×10^1 等表示。其中的 1.234×10^3 称为“规范化的指数形式”。即在字母 e(或 E)之前的小数部分中，小数点左边有且只有一位非零的数字。对于很大或很小的浮点数，用指数形式表示就尤为方便。例如 1230000000 可以用 1.23e9 表示，0.00000000123 可以用 1.23e−9 表示等。

2. 浮点型数据在内存中的存放形式

在 Python 2.7 解析器中，一个浮点型数据在内存中占 16 个字节。与整型数据的存



放方式不同,浮点型数据是按指数形式存储的。Python 解析器把一个浮点型数据分成小数部分和指数部分,分别进行存放。存放时是采用近似规范化的指数形式。例如浮点数 314.159,在内存中分配给该数值的存储单元的小数部分存放.314159,指数部分存放的是 3,同时,在一小段存储单元中存放该数值的符号位。实际上在计算机中是用二进制数来表示小数部分,用 2 的幂次来表示指数部分的。

3. 浮点型数据的舍入误差

由于浮点型变量在计算机内部存储的方式与整型变量不同,由此可能会产生一些误差,例如,我们都知道一个数 a 加上 10,其结果肯定比 a 大。但对于浮点数在计算机中进行运算时就可能会有所不同。请分析下面的程序:

```
#例 2-2  浮点型数据的舍入误差
>>>a=1234.567e10
>>>b=a+10
>>>print a
1.234567e+13
>>>print b
1.234567e+13
```

从输出的结果可以看到 a 和 b 的值是相同的,原因是:a 的值比 10 大得多,a+10 的理论值应是 12345670000010,而一个浮点型变量,当以指数形式输出时,它的最大有效数字位数只能是参加运算的变量中最大有效数字的位数,例如 a 和 10 相加运算,其中的最大有效数字是 a 的位数 7,所以输出的结果的最大有效数字是 7 位,后面的数是无意义的,因此并不能准确地表示该数。所以,在编写含有浮点型数据的程序时,应当避免将一个很大的数和一个很小的数直接相加或相减,否则就会“丢失”小的数。

2.1.4 布尔型

布尔型数据有两个布尔值,分别是 true(真)和 false(假),但只能取其中的一个。对于整型或浮点型,0 取 false,非 0 取 true;对于其他类型,空(或 NUll)取 false,非空取 true。布尔型数据最常用在条件语句或循环语句判断条件中,这将在第 3 章中讲解。

```
#例 2-3  布尔型数据
>>>print bool(0)
false
>>>print bool(1)
true
>>>print bool(0.0)
false
>>>print bool(0.1)
true
>>>print bool("")                    #空字符串
false
>>>print bool("Python")              #非空字符串
```

```
true
```

这里调用了 Python 内建的 bool()函数,把括号里面的内容转换成布尔类型的数据。

2.2 输入与输出

所谓的输入输出是以计算机主机为主体而言的。从计算机向外部输出(如显示器、打印机等)输出数据称为输出,从输入设备(如键盘、鼠标等)向计算机输入数据称为输入。

对于输出,Python 2.7 本身含有用于输出的 print 语句(Python 2.6 和 Python 3.x 系列则没有,它是由 Python 内建的 print 函数实现的)。对于输入,则是通过 Python 内建的 input 函数以及 raw_input 函数实现的。

2.2.1 print 语句

在前面的章节中已用到了 print 语句,它的作用是向终端(或系统隐含指定的输出设备)输出若干个任意类型的数据。Python 中的输出语句 print 和 C 语言中的 printf 函数很相似。

print 语句的一般格式为:

```
print 格式控制% (输出列表)
```

例如:

```
print "%d,%s" % (i,str)
```

格式控制是用双引号括起来的字符串,它包含两种信息:

(1) 格式声明。格式说明由“%”和格式字符组成,如%d、%s 等。它的作用是将输出的数据转换为指定的格式输出。格式声明总是由“%”字符开始的。

(2) 普通字符。普通字符即需要原样输出的字符。这样可以输出一些辅助信息,使输出的结果更清晰易读。

输出列表是需要输出的一些数据。

1. 格式字符

在输出时,对不同类型的数据要使用不同的格式字符。常用的有以下几种格式字符。

(1) d 格式符。用于输出十进制整数。有以下两种用法:

① d。按十进制整数数据的实际长度输出。

② %md。m 为指定的输出字段的宽度。如果数据的位数小于 m,则左端补空格;若大于 m,则按实际长度输出。

```
#例 2-4  print 输出语句中的 d 格式符
>>>a=12
>>>b=1234
>>>print "%3d,%3d" % (a,b)
 12,1234                              #注意,12 前面空了一格
```

(2) o格式符。以有符号八进制整数形式输出。注意,这里和C语言不同,C语言是以无符号八进制整数形式输出的。

```
#例2-5  print输出语句中的o格式符
>>>a=12
>>>b=-12
>>>print "%o" % (a)
14                              #输出八进制整数14
>>>print "%o" % (b)
-14                             #输出负的八进制整数-14
```

(3) x格式符。以有符号十六进制整数形式输出。若是小写的x,则输出的十六进制数;如果含有字母,则以小写形式显示;若是大写的X,则输出的十六进制数中的字母以大写形式显示。

```
#例2-6  print输出语句中的x(X)格式符
>>>a=12
>>>b=-12
>>>print "%x" % (a)
c                               #输出十六进制整数c(小写)
>>>print "%X" % (b)
-C                              #输出负的十六进制整数-C(大写)
```

(4) c格式符。用来输出一个字符。对于一个在0～255范围内的整数,也可以用“%c”使之按字符形式输出,在输出前,系统会将该整型数作为ASCII码转换成对应的字符。

```
#例2-7  print输出语句中的c格式符
>>>a=65
>>>b="A"
>>>print "%c" % (a)
A                               #输出ASCII码的十进制数65对应的字符"A"
>>>print "%c" % (b)
A                               #输出字符"A"
```

(5) s格式符。用来输出一个字符串。有以下几种用法:

① %s。按实际长度输出字符串。

② %(-)ms。输出的字符串占m列,如果字符串本身长度大于m,则突破m的限制,将字符串全部输出。若串长度小于m,则左边(右边)补空格。

③ %(-)m.ns。输出的字符串占m列,但只取字符串中左端n个字符。这n个字符输出在m列的右侧(左侧),左侧(右侧)补空格。如果n>m,则m自动取n值,保证n个字符能够正常输出。

```
#例2-8  print输出语句中的s格式符
>>>a="python"
>>>print "%s,%10s" % (a,a)
```

```
python,     python                 #输出的第二个 python 左边有 4 个空格
>>>print "%-10s" % (a)
python                             #python 右边有 4 个空格
>>>print "%10.5s" % (a)
      pytho                        #输出 python 中的前 5 个字符 pytho,左边有 5 个空格
>>>print "%-10.5s" % (a)
pytho                              #输出 python 中的前 5 个字符 pytho,右边有 5 个空格
```

(6) f 格式符。以小数形式输出浮点数。有以下两种用法:

① %f。不指定字段宽度,由系统自动指定,使整数部分全部输出,并输出 6 位小数。

② %(-)m.nf。输出的小数占 m 列,其中有 n 位小数。如果数值的长度小于 m,则左侧(右侧)补空格。

```
#例 2-9  print 输出语句中的 f 格式符
>>>a=1.111111111
>>>print "%f" % (a)
1.111111                           #输出的数值有 6 位小数
>>>print "%10.3f" % (a)
      1.111                        #输出的数值占 10 列,有 3 位小数,左边有 5 个空格
>>>print "%-10.3f" % (a)
1.111                              #输出的数值占 10 列,有 3 位小数,右边有 5 个空格
```

(7) e 格式符。以指数形式输出浮点数。有以下两种用法:

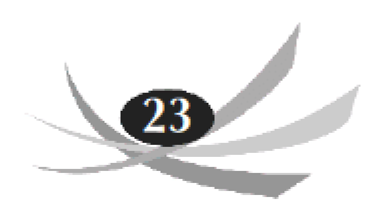
① %e。不指定字段宽度,有的系统自动指定输出的数据的小数位数为 6 位,指数部分占 4 位,其中"e"占 1 位,指数符号占 1 位,指数占 2 位。数值按规范化指数形式输出(即小数点前有且只有 1 位非零数字)

② %(-)m.ne。这里的 m、m、"-"的含义和(6)讲到的相同。

```
#例 2-10  print 输出语句中的 e 格式符
>>>a=111111111
>>>print "%e" % (a)
1.111111e+08                       #输出的数值有 6 位小数,指数部分占 4 位
>>>print "%10.3e" % (a)
 1.111e+08                         #输出的数值占 10 列,有 3 位小数,左边有 1 个空格
>>>print "%-10.3e" % (a)
1.111e+08                          #输出的数值占 10 列,有 3 位小数,右边有 1 个空格
```

(8) g 格式符。用来输出浮点数,它根据数值的大小,自动选 f 格式或 e 格式(选择时占宽度较小的一种),且不输出无意义的零,最多输出 6 位有效数字时,当数值的有效数字位数大于 6 位时,会对第 7 位数字进行四舍五入。

```
#例 2-11  print 输出语句中的 g 格式符
>>>a=123
>>>print "%f,%e,%g" % (a,a,a)
123.000000,1.230000e+02,123        #用%g 格式时,选择%f 格式输出
```




```
>>>a=123456789
>>>print "%f,%e,%g" % (a,a,a)
123456789.000000,1.234568e+08,1.23457e+08                #用%g格式时,选择%e格式输出
```

当 a＝123 时,用%f 格式输出占 10 列,用%e 格式输出时占 12 列,用%g 格式时,自动从上面两种格式中选择较短的一种(今以%f 格式为短),故占 10 列,并按%f 格式用小数输出,最后 6 位小数位为无意义的 0,不输出,因此用%g 格式最终输出 123,占 3 列。当 a＝123456789 时,此时数值的有效数字位数大于 6 位,用%f 格式输出占 16 列,用%e 格式输出时占 12 列,用%g 格式时,自动从上面两种格式中选择较短的一种(今以%e 格式为短),故占 12 列,并按%e 格式用指数输出,此时数值的有效数字位数大于 6 位,对第 7 位数字进行四舍五入,输出数值的有效数字达到最大值 6 位,因此用%g 格式最终输出 1.23457e＋08,占 11 列。

以上介绍了 8 种格式符,归纳如表 2-1 所示。

表 2-1　print 语句的格式字符

格式字符	说　　明
d	以带符号的十进制形式输出整数(正数不输出符号)
o	以有符号八进制整数形式输出(不输出前导符 0)
x,X	以有符号十六进制整数形式输出(不输出前导符 0x),若是小写的 x,则输出的十六进制数中如果含有字母,则以小写形式显示;若是大写的 X,则输出的十六进制数中的字母以大写形式显示
c	以字符形式输出一个字符
s	输出一个字符串
f	以小数形式输出浮点数,并输出 6 位小数
e,E	以指数形式输出浮点数,若是小写的 e,则输出的指数字母也为小写的 e,若是大写的 E,则输出的指数字母也为大写的 E
g,G	用来输出浮点数,它根据数值的大小,自动选 f 格式或 e 格式(选择时占宽度较小的一种),且不输出无意义的零,最多输出 6 位有效数字时,当数值的有效数字位数大于 6 位时,会对第 7 位数字进行四舍五入

在格式说明中,在%和上述格式字符间可以输入以下几种附加符号,见表 2-2。

表 2-2　print 语句的附加格式字符

字　　符	说　　明
m(代表一个正整数)	数据的最小宽度
n(代表一个正整数)	若是实数,则表示输出 n 个小数;若是字符串,则表示截取的字符个数
—	输出的数据向左靠

2.2.2　input 函数与 raw_input 函数

有时候往往需要用户向程序中输入数据,这时就需要用到 Python 提供的用于输入

的函数。Python 的输入比 C 语言的输入要简单一些，在 python 的输入中，提供了两个内建的函数 input()和 raw_input()函数。input()函数是把用户输入的数据处理成合法的 Python 表达式，而 raw_input()函数是把用户输入的数据处理成字符串。此外，这两个函数也可以使用前一小节介绍的格式字符，以得到想要的输入提示。关于 input()函数的用法请看例 2-12。

```
#例 2-12   input()函数的使用
>>>language=input("which programming language do you like?")
which programming language do you like?Python
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
  File "<string>", line 1, in<module>
NameError: name 'Python' is not defined          #此时提示 Python 未定义
>>>language=input("which programming language do you like?")
which programming language do you like?"Python"
>>>print language
Python                                           #此时程序可以正常运行
>>>type(language)
<type 'str'>                                     #language 为字符串类型
>>>number=input("what's your favorite number?")
what's your favorite number?11
type(number)
<type 'int'>                                     #number 为整型
>>>num1,num2=3,5
>>>answer=input("%d+%d=" % (num1,num2))
3+5=                                             #使用格式字符以更好地提示用户输入
```

从上面程序可以看出，当第一次输入 Python 时，由于没有用引号括起来，就不是字符串，但 Python 也不是一个合法的数据类型，所以会报错，提示 Python 未定义，当第二次输入“Python”时，此时有引号括起来，也就是说输入的是字符串，于是把“Python”字符串赋给 language 变量，再用 print 语句输出 language 时就能正确输出字符串 Python 了。使用 Python 内建的 type()函数可以求出变量的类型，通过 type(language)可以得到变量 language 的类型为字符串类型，通过 type(number)可以得到变量 number 的类型为整型。当使用格式字符时，它们会被相应的参数值所替换，如上面的“%d”分别被 num1 和 num2 的值所替换，所以就输出“3+5=”，然后等待用户的输入。

下面通过例 2-13，掌握 raw_input()函数的用法。

```
#例 2-13   raw_input()函数的使用
>>>language=raw_input("which programming language do you like?")
which programming language do you like?Python
>>>print language
Python                              #注意，输入 Python 时没有用引号括起来，程序可以正常运行
>>>type(language)
```

```
<type 'str'>                    #language 为字符串类型
>>>number=raw_input("what's your favorite number?")
what's your favorite number?11
type(number)
<type 'str'>                    #和用 input 函数不同的是,此时的 number 为字符串类型
```

通过这两个例子,我们可以发现:使用 input()函数输入的数据是 Python 合法的表达式,而用 raw_input()函数输入的数据都将转换为字符串类型。所以,为输入的简便,程序在无特殊要求的情况下都使用 raw_input()函数来接收用户输入的数据。

2.3　运　算　符

2.3.1　Python 语言运算符简介

Python 语言的运算符主要有以下几类:

(1) 算术运算符(+、-、*、/、%、**、//);

(2) 关系运算符(>、<、==、>=、<=、!=、<>);

(3) 逻辑运算符(and、or、not);

(4) 位运算符(&、|、^、~、<<、>>);

(5) 赋值运算符(=以及复合赋值运算符);

(6) 成员运算符(in、not in);

(7) 同一运算符(is、is not)。

下面各小节将对这些运算符进行讲解。

2.3.2　算术运算符和算术表达式

1. 基本的算术运算符

(1) +(加法运算符,或正值运算符,如 2+3、+2)。

(2) -(减法运算符,或负值运算符,如 3-2、-3)。

(3) *(乘法运算符,如 2*3)。

(4) /(除法运算符,如 3/2)。

(5) %(取模运算符或称求余运算符)。

(6) **(幂运算符)。

(7) //(取整除运算符,主要用于浮点数,结果为小于商的最大整数)。

```
#例 2.14　算术运算符的使用
>>>a=2+3
>>>b=3-2
>>>c=2*3
>>>d=7/2
>>>e=7/2.0
>>>f=7%2
```

```
>>>g=7%2.3
>>>h=2**3
>>>i=3//2.0
>>>print "a=%d,b=%d,c=%d,d=%d,e=%g,f=%d,g=%g,h=%d,i=%g" % (a,b,c,d,e,f,g,h,i)
a=5,b=1,c=6,d=3,e=3.5,f=1,g=0.1,h=8,i=1
```

这里需要说明的是,对于单斜杠(/)除法运算符,如果相除的两个数都是整数,则相除的结果是小于商的最大整数,如 7/2=3。如果相除的两个数中有一个或两个都是浮点数,则返回的结果就是商,如 7/2.0=3.5。对于双斜杠(//)除法运算符,无论相除的两个数是整数还是浮点数,其结果都是小于商的最大整数。同时还要注意的是,Python 的求余运算符不同于 C 语言的求余运算符,C 语言的求余运算符只支持对整数的求余,而 Python 的求余运算符支持浮点数的求余,如 7%2.3=0.1。

2. 算术表达式和运算符的优先级

用算术运算符和括号将运算对象连接起来的,符合 Python 语法规则的式子,称为 Python 算术表达式。运算对象包括常量、变量、函数等。

```
#例 2-15  算术表达式
>>>a=5
>>>b=3
>>>c=2
>>>print a+3*b/2-2.5
6.5
```

Python 语言规定了运算符的优先级。在表达式求值时,按照运算符的优先级别高低次序执行,例如先乘除后加减。如例 2-15 中的表达式 a+3*b/2-2.5,3 的左侧为加号,右侧为乘号,而乘号优先级高于减号,因此,相当于 a+(3*b)/2-2.5,同理,除号的优先级比加号高,表达式 a+3*b/2-2.5 又相当于 a+((3*b)/2)-2.5。所以,表达式 a+3*b/2-2.5 算出的结果为 6.5。

3. 强制类型转换函数

与 C 语言不同,C 语言有强制类型转换符,其形式是(类型名)(表达式),Python 语言没有强制类型转换符,但 Python 语言提供了强制类型转换的函数,如前面提到的将某一类型转换成布尔型的 bool()函数,还有转换成整型的 int()函数、转换成浮点型的 float()函数等。需要说明的是,在使用强制类型转换函数时,得到一个所需类型的中间变量,原来变量的类型未发生变化。

```
#例 2-16  强制类型转换函数
>>>a=3.5
>>>b=int(a)
>>>print "a=%f,b=%d" % (a,b)
a=3.500000,b=3                    #a 的类型仍为 float 类型,值仍等于 3.5
```

2.3.3 关系运算符和关系表达式

所谓关系运算实际上是比较运算。将两个值进行比较,判断其比较的结果是否符合

给定的条件。例如 a>5 是一个关系表达式，大于号(>)是一个关系运算符，如果 a=3，则不满足给定的“a>5”条件，因此关系表达式的值为“假”；如果 a=7，则满足“a>5”条件，所以关系表达式的值为“真”。

1. 关系运算符及其优先级

Python 语言提供 7 种关系运算符：

(1) >(大于)；

(2) >=(大于或等于)；

(3) <(小于)；

(4) <=(小于或等于)；

(5) ==(等于)；

(6) !=(不等于)；

(7) <>(不等于，和“!=”功能一样)。

关于优先级别：

(1) 在 Python 语言中，所有的关系运算符的优先级别都相同，但在 C 语言中，前 4 种关系运算符(>、>=、<、<=)的优先级别相同，后两种也相同。前 4 种高于后 2 种。这一点需要注意。

(2) 关系运算符的优先级低于算术运算符。

(3) 关系运算符的优先级高于赋值运算符。

例如：a>b+c 等效于 a>(b+c)；a=b>c 等效于 a=(b>c)。

```
#例 2-17　关系运算符的使用
>>>a=3
>>>b=5
>>>c=7
>>>a>b
false
>>>a<b
true
>>>a>=b
false
>>>a<=b
true
>>>a==b
false
>>>a==3
true
>>>a!=b
true
>>>a<>b
true
>>>a!=b>c
```

```
false                           #3!=5,返回 true,true将自动转为 1,然后判断 1>7,所以返回
                                #false,在 C语言中将返回 1,即 true(C语言中,大于运算符优
                                #先级高于不等于运算符)
>>>a>b+c                        #相当于 3>(5+7),不满足条件,所以返回 false
false
>>>a=b>c
>>>print a
false                           #相当于 a=(5>7),5>7返回 false,再赋给 a,所以输出 false
```

2. 关系表达式

用关系运算符将两个表达式(可以是算术表达式、关系表达式、逻辑表达式、字符串表达式)连接起来的式子称为关系表达式。例如,下面列出的几个表达式都是合法的关系表达式:a>b、a+b>c+d、(a>b)>(c>d)、“abc”>“bcd”

关系表达式的值是一个逻辑值,即 true 或 false。例如,关系表达式“5>3”的值为 true。

```
#例 2-18  关系表达式
>>>a=3
>>>b=5
>>>c=7
>>>d=9
>>>a+b>c+d
false                           #相当于(3+5)>(7+9),即 8>16,所以返回 false
>>>(a>b)>(c>d)
false                           #相当于(3>5)>(7>9),即 0>0,所以返回 false
>>>"abc">"acd"
false                           #从第一个字符开始比较,此时比较的是字符的 ASCII码
                                #第一个字符相同,比较下一个字符,b的 ASCII码小于 c
                                #的 ASCII码,所以返回 false
```

2.3.4 逻辑运算符和逻辑表达式

用逻辑运算符将关系表达式或逻辑量连接起来的式子称为逻辑表达式。在 C 语言中有以下形式的逻辑表达式:

```
(a>b)&&(c>d)
```

如果 a>b 且 c>d,则上述逻辑表达式的值为“真”。下面介绍 Python 语言中的逻辑运算符和逻辑表达式。

1. 逻辑运算符及其优先级

Python 语言提供三种逻辑运算符:

(1) and 逻辑与(相当于 C 语言中的 &&);

(2) or 逻辑或(相当于 C 语言中的 ‖);

(3) not 逻辑非(相当于 C 语言中的!)。

and 和 or 是双目运算符,它要求有两个运算量,如(a>b)and(c>d),(a>b)or(c>

d)。not是一目运算符,只要求有一个运算量,如not(a>b)。

表2-3为逻辑运算的真值表。用它来表示当x和y的值为不同的组成时,各种逻辑运算所得到的值。

表2-3 逻辑运算的真值表

x	y	not x	not y	x and y	x or y
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

在一个逻辑表达式中如果包含多个逻辑运算符,例如:

```
not(a>b) and (b>c) or d
```

按以下的有效次序:

(1) not(非)→and(与)→or(或);

(2) 逻辑运算符低于关系运算符,但对于C语言则有些不同,在C语言中,逻辑运算符中的not(非)高于算术运算符(即也高于关系运算符)。

例如:

a>b and c>d 相当于(a>b) and (c>d);

not a+b>c 相当于 not ((a+b)>c),在C语言则相当于((! a)+b)>c。

```
#例2-19  逻辑运算符的使用1
>>>x=true
>>>y=false
>>>x and y
false
>>>x and x
true
>>>x or y
true
>>>not x
false
>>>not y
true
>>>1>2 and 3>2
false                        #相当于false and true,所以返回false
>>>not 2+3>1                 #相当于not((2+3)>1),即not(5>1),返回false
false                        #在C语言中相当于((not 2)+3)>1,返回1(即true)
```

2. 逻辑表达式

如前所述,逻辑表达式就是用逻辑运算符将关系表达式或逻辑量连接起来的式子。

Python 的逻辑表达式的值可以是布尔值(true、false),也可以是整数、浮点数,甚至还可以是字符串。这与C语言的逻辑表达式的值不同,在C语言中,逻辑表达式的值只能是一个逻辑量1(代表"真")或0(代表"假")。但在判断一个量是否为"真"时,Python 和C语言却是一样,都是以0或空字符(串)代表"假",以非零或非空字符(串)代表"真"。

```
#例 2-20  逻辑运算符的使用 2
>>>a=3
>>>b=5
>>>c=4.5
>>>d="C"
>>>e="Python"
>>>not a
false                             #对 3(true)取非,所以返回 false
>>>a and b
5                                 #a,b都为 true,返回最后一个表达式的值,即 5(b)
>>>b and a
3                                 #a,b都为 true,返回最后一个表达式的值,即 3(a)
>>>a or 0
3                                 #a为 true,0为 false,所以,整个表达式返回 a的值 3
>>>b and c
4.5                               #b,c都为 true,返回最后一个表达式的值,即 4.5(c)
>>>d and e
"Python"                          #d,e都为 true,返回最后一个表达式的值,即"Python"(e)
```

熟练掌握 Python 语言的关系运算符和逻辑运算符之后,可以巧妙地用一个逻辑表达式来表示一个复杂的条件。

例如,要判别用 year 变量表示的某一年是否为闰年。如果是闰年,该年份要么能被4整除,但又不能被100整除,如2016;要么能被400整除,如2000。可以用一个逻辑表达式来表示:

```
((year%4==0 and year%100!=0) or (year%400==0))
```

如果判断非闰年,只要在上面的表达式前面取非,即:

```
not ((year%4==0 and year%100!=0) or (year%400==0))
```

2.3.5 位运算符

1. 位运算符和位运算

所谓位运算是指进行二进制位的运算,Python 语言提供表2-4所列出的位运算符。

表 2-4 位 运 算 符

运 算 符	含 义	运 算 符	含 义
&	按位与	~	取反

续表

运 算 符	含 义	运 算 符	含 义
|	按位或	<<	左移
^	按位异或	>>	右移

说明：

(1) 位运算符中除～以外，其他均为二目运算符，即要求两侧各有一个运算量。

(2) 运算量只能是整型数据，不能为浮点型或字符串型。

下面对各运算符分别进行介绍。

2. “按位与”运算符(&)

参加运算的两个整型数据，按二进制位进行“与”运算。如果两个相应的二进制位都为1，则该位的结果为1，否则为0。即：

```
0&0=0,    0&1=0,    1&0=0,    1&1=1
```

为简单起见，这一小节讲解的十进制数都假设用一个字节来表示。

例如，5&9，即：

```
      0 0 0 0 0 1 0 1        (5)
(&)   0 0 0 0 1 0 0 1        (9)
-------------------------------
      0 0 0 0 0 0 0 1        (1)
```

因此，5&9 的值为1。注意，如果参加 & 运算的是负数，则以补码形式表示为二进制数，然后再按位进行“与”运算。

按位与有一些特殊用途：

(1) 清零。如果想将数清零，即使得其全部的二进制位为0，只需将该数与0进行与运算，即可达到清零的目的。

例如，将数25(00011001)清零，只需将其与0进行与运算即可，即：

```
      0 0 0 1 1 0 0 1        (25)
(&)   0 0 0 0 0 0 0 0        (0)
-------------------------------
      0 0 0 0 0 0 0 0        (0)
```

(2) 取一个数中某些指定位或者保留某些指定位。可以通过将这个数与另外一个数(这个数在要取指定位的位置上是1)进行与运算即可达到目的。

例如，取数25的低4位的值，只需将其与15(00001111，即对应的位为1)进行与运算即可，即：

```
      0 0 0 1 1 0 0 1        (25)
(&)   0 0 0 0 1 1 1 1        (15)
-------------------------------
      0 0 0 0 1 0 0 1        (9)
```

3. “按位或”运算符(|)

参加运算的两个整型数据，按二进制位进行“或”运算。如果两个相应的二进制位都为0，则该位的结果为0，否则为1。即：

```
0|0=0,     0|1=1,     1|0=1,     1|1=1
```

例如,5|9,这里假设用一个字节来表示,即:

```
        0 0 0 0 0 1 0 1        (5)
(|)     0 0 0 0 1 0 0 1        (9)
--------------------------------------
        0 0 0 0 1 1 0 1        (13)
```

因此,5|9 的值为 13。

按位或运算常用来对一个数据的某些位设置为 1。

例如,把数 75 的低 4 位的值设置为 1,只需将其与 15(00001111,即对应的位为 1)进行或运算即可,即:

```
        0 1 0 0 1 0 1 1        (75)
(|)     0 0 0 0 1 1 1 1        (15)
--------------------------------------
        0 1 0 0 1 1 1 1        (79)
```

所以 75|15=79,低 4 位为 1,高 4 位保留原样。

4. “按位异或”运算符(^)

参加运算的两个整型数据,按二进制位进行“异或”运算。如果两个相应的二进制位同号,则该位的结果为 0,否则为 1。即:

```
0^0=0,     0^1=1,     1^0=1,     1^1=0
```

例如,5^9,这里假设用一个字节来表示,即:

```
        0 0 0 0 0 1 0 1        (5)
(^)     0 0 0 0 1 0 0 1        (9)
--------------------------------------
        0 0 0 0 1 1 0 0        (12)
```

因此,5^9 的值为 12。

“异或”的意思是判断两个相应的位值是否相“异”,若“异”(即值不同)则为 1,否则为 0。

按位异或有一些特殊用途:

(1) 使特定位翻转。如果想将数的某些特定位翻转,即 1 变为 0,0 变为 1。只需将该数与另外一个数(这个数在要翻转时指定位的位置上是 1)进行异或运算即可达到目的。

例如,把数 75 的低 4 位翻转,只需将其与 15(00001111,即对应的位为 1)进行异或运算即可,即:

```
        0 1 0 0 1 0 1 1        (75)
(^)     0 0 0 0 1 1 1 1        (15)
--------------------------------------
        0 1 0 0 0 1 0 0        (68)
```

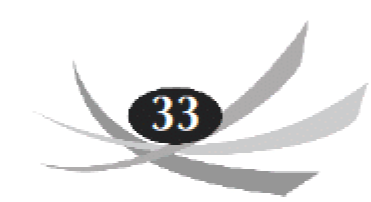
结果值的低 4 位正好是原来的数低 4 位的翻转,高 4 位保留原样。

(2) 与 0 相异或,保留原值。

从上面 75 与 15 相异或的例子可以看出,其结果值的高 4 位保留原样。这是因为原数种的 1 与 0 进行异或运算得 1,0^0 得 0,所以保留原样。

(3) 交换两个值,不用临时变量。

例如,a=5,b=9。将 a 和 b 的值互换而不用临时变量。可以用以下赋值语句实现:




```
a=a^b
b=a^b
a=a^b
```

具体是如何实现的呢？请看下面的分析：

```
       0 0 0 0 0 1 0 1        (a=5)
(^)    0 0 0 0 1 0 0 1        (b=9)
--------------------------------------
       0 0 0 0 1 1 0 0        (a^b的结果赋给a,所以a=12)
(^)    0 0 0 0 1 0 0 1        (b=9)
--------------------------------------
       0 0 0 0 0 1 0 1        (a^b的结果赋给b,此时b=5)
(^)    0 0 0 0 1 1 0 0        (a=12)
--------------------------------------
       0 0 0 0 1 0 0 1        (a^b的结果赋给a,此时a=9)
```

即执行第二条语句 b=a^b 时，实际上是相当于执行 b=(a^b)^b。又由于(a^b)^b 等于 a^b^b，且 b^b 的结果为 0，所以，有 b=(a^b)^b=a^0=a=5。执行第三条语句 a=a^b 时，实际上是相当于执行(a^b)^((a^b)^b)，所以，有 a=(a^b)^((a^b)^b)=a^a^b^b^b=0^0^b=b=9。

5. "取反"运算符(～)

～是一个单目运算符，用来对一个二进制数按位取反，即将 0 变 1，1 变 0。

例如，将十进制数 9 按位取反，即：

```
(～)  0 0 0 0 1 0 0 1        (9)
------------------------------------
      1 1 1 1 0 1 1 0        (-10)
```

注意，对 9 取反后得到的结果是一个负数，根据补码的知识，可以算出对 9 取反后得到的结果(11110110)是−10 的补码。

下面举例说明取反运算符的应用。

如果想要使一个整数 a(这里假设 a 为 9)的最低一位为 0，可以对 1 取反，再和 a 进行与运算，a=a&～1 即：

```
(～)  0 0 0 0 0 0 0 1        (1)
------------------------------------
      1 1 1 1 1 1 1 0
(&)   0 0 0 0 1 0 0 1        (9)
------------------------------------
      0 0 0 0 1 0 0 0        (8)
```

～运算符的优先级别比算术运算符、关系运算符、逻辑运算符和其他位运算符都高。

6. 左移运算符(<<)

左移运算符用来将一个数的二进制位全部左移若干位。高位左移后溢出，舍弃，低位补零。例如：

$$a=a<<3$$

将 a 的二进制数左移 3 位，右(即低位)补 0。若 a=9，即二进制数 00001001，左移 3 位得到 01001000，即十进制数 72。

当一个数左移时被溢出舍弃的高位中不包含 1 时，左移 1 位相当于该数乘以 2，左移 n 位相当于该数乘以 2^n。上面举的例子 $9<<3=72$，即乘以 8(2 的 3 次方)。

7. 右移运算符(>>)

右移运算符用来将一个数的二进制位全部右移若干位。低位右移后溢出,舍弃,高位补符号位,即正数,高位补 0,负数,高位则补 1。例如:

$$a=a>>3$$

若 a=72,将 a 的二进制数右移 3 位,正数,左(即高位)补 0。即二进制数 01001000,右移 3 位得到 00001001,即十进制数 9。

当一个数右移时被溢出舍弃的低位中不包含 1 时,右移 1 位相当于该数除以 2,右移 n 位相当于该数除以 2^n。上面举的例子 72>>3=9,即除以 8(2 的 3 次方)。

```
#例 2-21  位运算符的使用
>>>a=5
>>>b=9

#与运算(&)
>>>a&b
1                                   #将 a 和 b 进行与运算得 1
>>>25&0
0                                   #将 25 的所有二进制位清零
>>>25&15
9                                   #取 25 的低 4 位的值

#或运算(|)
>>>a|b
13                                  #将 a 和 b 进行或运算得 13
>>>75|15
79                                  #将 75 的低 4 位的值设置为 1

#异或运算(^)
>>>a^b
12                                  #将 a 和 b 进行异或运算得 12
>>>75^15
68                                  #将 75 的低 4 位翻转
>>>a=a^b
>>>b=a^b
>>>a=a^b
>>>print "a=%d,b=%d" % (a,b)
a=9,b=5                             #不用中间变量,将 a、b 的值交换

#取反运算(~)
>>>~a
-10                                 #将 a(此时 a=9)取反得到-10
>>>a&~1
8
```

```
#左移运算(<<)
>>>a<<3
72                                    #将 a 左移 3 位得到 72

#右移运算(>>)
>>>72>>3
9                                     #将 72 右移 3 位得到 9
```

2.3.6　赋值运算符

1. 赋值运算符

赋值符号“＝”就是赋值运算符，它的作用是将一个数据或者是表达式赋给一个变量。如“a＝5”的作用是执行一次赋值操作，或者称为赋值运算。

2. 复合赋值运算符

在赋值符“＝”之前加上其他的运算符，可以构成复合的运算符，或者称为复合赋值运算符。如果在“＝”前加上一个“＋”运算符就构成了复合赋值运算符“＋＝”。例如：

a＋＝5 相当于 a＝a＋5；

a/＝a－3 相当于 a＝a/(a－3)；

a&＝3 相当于 a＝a&3。

对于复合赋值运算符，为便于记忆，我们可以这样理解：

(1) a＋＝b(其中 a 为变量，b 为表达式)；

(2) a＋＝b(将有下划线的“a＋”移到“＝”右侧)；

(3) a＝a＋b(在“＝”左侧补上变量名 a)。

注意，b 是一个表达式，所以在做运算时，它相当于有括号。

凡是二目运算符，都可以与赋值符一起组成复合赋值运算符。例如＋＝、－＝、*＝、/＝、<<＝、>>＝、&＝、|＝等。

```
#例 2-22　复合赋值运算符的使用
>>>a=3
>>>a+=5
>>>print a
8                                     #相当于 a=a+5,a=3,所以输出运算结果得到 8
>>>a /=a -3
>>>print a
1                                     #相当于 a=a /(a-3),a=8,所以输出运算结果得到 1
>>>a &=3
>>>print a
1                                     #相当于 a=a&3,a=1,所以输出运算结果得到 1
```

2.3.7　其他运算符

除了以上的一些运算符之外，Python 还支持成员运算符和同一运算符，本小节主要

讲解这两个运算符。

1. 成员运算符

成员运算符(in)的作用是用来判断某个数据或者变量是否在给定的数据对象中,如果在,则返回 true,否则返回 false。而成员运算符(not in)的作用正好相反,即用来判断某个数据或者是变量是否不在给定的数据对象中,如果不在,则返回 true,否则返回 false。这里的数据对象可以是字符串、列表、元组、字典和集合。关于列表、元组、字典和集合将在第 4 章中介绍。例如:

"a" in "way",结果将返回 true,而"a" not in "way",结果将返回 false。

"b" in "way",结果将返回 false,而"b" not in "way",结果将返回 true。

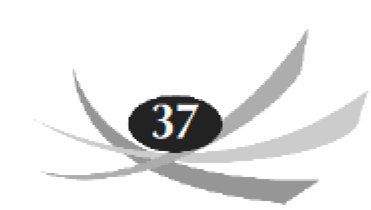
```
#例 2-23  成员运算符的使用
#给定的数据对象是字符串
>>>str="Python"
>>>"n" in str
true
>>>"e" in str
false
>>>"n" not in str
false
>>>"e" not in str
true

#给定的数据对象是列表
>>>a=1
>>>b=7
>>>list=[1,2,3,4,5]            #这里定义了一个列表变量 list
>>>a in list
true
>>>b in list
false
>>>a not in list
false
>>>b not in list
true
```

2. 同一运算符

同一运算符(is)的作用是用来判断两个标识符是否引用自同一个对象,如果是,则返回 true,否则返回 false。而同一运算符(is not)的作用正好相反,即用来判断两个标识符不是引用自同一个对象,如果不是,则返回 true,否则返回 false。

```
#例 2-24  同一运算符的使用
>>>a=3
```




```
>>>b=3
>>>c=3.5
>>>d=3.5
>>>id(a)                    #id()函数是求标识符的内存地址
39941952                    #这个值是随机分配的,不同的机器,不同的时间运行,其值也不一样
>>>id(b)
39941952
>>>a is b
true                        #a,b的内存地址相同,即它们引用自同一个对象,所以返回 true
>>>a is not b
false
>>>id(c)
39974792
>>>id(d)
39974776
>>>c is d
false                       #c,d的内存地址不同,它们不是引用同一个对象,所以返回 false
>>>c is not d
true
```

2.3.8　运算符的优先级

在前面各小节中也对一些运算符的优先级进行了介绍,为了让读者更好地掌握运算符的优先级,这里将对常用的运算符优先级进行归纳总结。表 2-5 列出了常用的运算符及它们的优先级次序。

表 2-5　常用的运算符及它们的优先级次序(从高到低依次排序)

运　算　符	描　　述
* *	指数
～x	按位取反
＋x,－x	取正、取负
*,/,//,%	乘法、除法、整数除与取余
＋,－	加法与减法
<<,>>	移位
&	按位与
^	按位异或
		按位或
<,<=,>,>=,!=,<>,==	关系(比较)
is,is not	同一性测试

续表

运　算　符	描　　述
in,not in	成员测试
not x	逻辑“非”
and	逻辑“与”
or	逻辑“或”

说明：

(1) 在这些常用的运算符中，指数运算符(* *)的优先级是最高的。例如，表达式 a+2 * * 3，则应先进行指数运算，求出 2 的 3 次幂，然后再与 a 进行加法运算，即 a+(2 * * 3)。

(2) 按位取反运算符(~)的优先级比算术运算符、关系运算符、逻辑运算符和其他位运算符都高，例如，~a&b，则先进行~a 运算，再与 b 进行与运算，即(~a)&b。

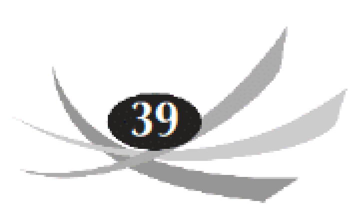
(3) 对于关系运算符，在 Python 语言中，所有的关系运算符的优先级别都相同，但在 C 语言中，关系运算符>、>=、<、<=的优先级别相同，!=，==这两种也相同。但前 4 种高于后 2 种。这一点需要注意。例如 0==2>3，在 Python 中相当于(0==2)>3，结果返回 false，但在 C 语言中则相当于 0==(2>3)，结果返回 true。

(4) 对于逻辑运算符，在 Python 语言中，所有的逻辑运算符都低于关系运算符，但对于 C 语言则有些不同，在 C 语言中，逻辑运算符中的 not(非)高于算术运算符(即也高于关系运算符)。例如 not 1+2，在 Python 中相当于 not (1+2)，结果返回 false，但在 C 语言中则相当于(not 1)+2(即(! 1)+2)，结果返回 2。

(5) 关系运算符的优先级低于算术运算符。例如 a>b+c，相当于 a>(b+c)。

(6) 可以通过括号来改变运算符的执行顺序。例如 1>2+3，没有括号时，执行顺序是先计算 2+3，求出结果 5，然后再与 1 作比较，得到结果为 false。但在加号前加上括号，即(1>2)+3，此时先执行括号里面的内容，得到 false，然后把 false 转为 0 再与 3 相加，得到结果 3。

```
#例 2-25  运算符优先级
>>>1+2* * 3
9                     #幂运算符的优先级比加法的高,先执行幂运算,再执行加法运算
>>> (1+2) * * 3
27
>>>~1&2
2                     #取反运算符的优先级比与运算的高,先执行取反运算,再执行与运算
>>>~ (1&2)
-1
>>>0==2>3
false                 #所有关系运算符的优先级相同,按从左到右的顺序执行
>>>0== (2>3)
true
>>>not 1+2
```




```
false                    #加法运算符的优先级比逻辑非的高,先执行加法运算,再执行逻辑非运算
>>> (not 1)+2
2
>>>1>2+3
false                    #算术运算符的优先级比关系运算符高,先执行算术运算,再执行关系运算
>>> (1>2)+3
3
```

2.4　本章小结

本章主要讲解了以下几个知识点：

(1) 什么是变量。变量是一种使用方便的占位符,用于引用计算机内存地址。在定义变量和选择其他标识符时,应该做到“见名知意”,即选择有含义的单词或其缩写作为标识符。同时还介绍了 Pascal Notation(帕斯卡命名法)、Camel Notation(驼峰命名法)以及 Hungarian Notation(匈牙利命名法)这三种最常用的标识符命名方法。

(2) Python 的数据类型。本章主要讲解 Python 的基本数据类型,包括整型、浮点型、字符串、布尔类型。整型数据可以分为整型和长整型。整数表示方法可以用十进制、八进制和十六进制表示。浮点数可以用十进制小数形式和指数形式表示。字符串类型数据是用单引号'、双引号"或三个引号'''括起来的任意字符。布尔型数据有两个布尔值,分别是 true(真)和 false(假)。布尔型数据最常用在条件语句或循环语句判断条件中。

(3) Python 的输入输出。Python 的输入是通过 print 语句实现的,同时还介绍了 8 种格式符,如用于输出十进制整数的 d 格式符;以小数形式输出浮点数的 f 格式符等。而输出则是通过 input 和 raw_input 函数实现的,input()函数是把用户输入的数据处理成合法的 Python 表达式,而 raw_input()函数是把用户输入的数据处理成字符串。

(4) Python 的运算符。主要包括算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符、成员运算符以及同一运算符。

(5) Python 运算符的优先级。逻辑运算符的优先级低于关系运算符,而关系运算符的优先级低于算术运算符。可以通过括号来改变运算符的执行顺序。

2.5　习　　题

一、解答题

1. 什么是变量,变量类型有哪些?
2. Python 有哪些输入函数? 它们有哪些区别?
3. Python 有哪些常用的运算符? 它们的优先级又是怎样的?

二、计算题

1. 请将下面各数用八进制和十六进制数表示：

(1) 13　　(2) 65　　(3) 129　　(4) 255

(5) －123　　(6) 781　　(7) －1024　　(8) 32678

2. 写出下面表达式的结果。

(1) 2＋3＊＊3/2－7　　(2) not 2＞1＞5＋4　　(3) 7|9&～2

(4) a＞b and a＞c or b＝＝c(a＝5,b＝3,c＝7)　　(5) 23＞＞2^9

三、上机练习

1. 从终端接收一个年份,判断该年是否是闰年。

2. 利用 Python 作为科学计算器。熟悉 Python 中的常用运算符,并分别求出表达式 11＊25//2＋16－789％10、11＊(25//(2＋16)－789)％10、75＞＞2＊2&12 的值。

第 3 章

程序流程控制

本章学习目标

- 理解算法的基本概论
- 掌握算法的表示方法
- 会利用“自上而下求精法”来求解问题
- 掌握 if 选择语句
- 掌握 while 和 for 循环语句
- 会使用 break 和 continue 语句控制程序的执行顺序

本章先向读者介绍一些有关算法的基本概论，什么是算法、算法的要素及特性。算法的表示方法和常用的结构化程序设计方法——自上而下求精法。然后介绍 if 选择语句以及它的基本变形，紧接着介绍 while 和 for 循环语句，同时还介绍了循环语句中经常用到的 break 和 continue 语句。最后，本章末尾给出的练习题将使读者进一步巩固本章重要的知识点。

3.1 算 法 概 述

3.1.1 算法及其要素和特性

1. 算法的定义

算法(Algorithm)是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出。如果一个算法有缺陷，或不适合于某个问题，执行这个算法将不会解决这个问题。不同的算法可能用不同的时间、空间或效率来完成同样的任务。一个算法的优劣可以用空间复杂度与时间复杂度来衡量。

“算法”这个词其实并不陌生，因为从小学开始，大家就接触算法了。例如，数学里面的四则运算，它必须要按照一定的算法步骤进行运算。“先运算括号里面的内容，再运算括号外的，先乘除后加减”，其实，这就是四则运算的算法。在中学学习到的指数运算、矩阵运算和代数运算等都可以说是一种算法。

2. 算法的三要素

算法的三要素包括操作、控制结构、数据结构。

(1) 操作。尽管算法的实现平台有许多种类,它们的函数库、类库也有较大的差异,但算法所必须具备的最基本的操作功能是相同的。这些操作包括以下几个方面。

算术运算:加、减、乘、除。

关系运算:大于、小于、等于、不等于。

逻辑运算:与、或、非。

数据传输:输入、输出(计算)、赋值(计算)。

(2) 算法的控制结构。一个算法功能的实现不仅取决于所选用的操作,还取决于各操作之间的执行顺序,即控制结构。算法的控制结构给出了算法的框架,决定了各操作的执行次序。这些结构包括以下几个方面。

顺序结构:各操作是依次执行的。

选择结构:根据条件是否成立来决定选择执行哪个操作。

循环结构:有些操作要重复执行,直到满足一定的条件时才结束,这种控制结构也称为重复或迭代结构。

(3) 数据结构。算法操作的对象是数据,数据间的逻辑关系、数据的存储方式以及处理方式就是数据结构。

3. 算法的基本特征

一个正确、有价值的算法必须具有以下5个特性。

(1) 有穷性。一个算法应包含有限的操作步骤,而不能是无限的。实际上,“有穷性”往往是指在合理的范围内,如果让计算机执行一个历时100年才结束的算法,这虽然是有穷的,但超过了合理的限度,人们也不把它视为有穷的算法。

(2) 确定性。算法中的每一个步骤都应当是确定的,而不应当是模糊的。算法中的每一个步骤应当不致被解释成不同的含义,而应是十分明确无误的。

(3) 有零个或多个输入。一个算法有零个或多个输入,以刻画运算对象的初始情况,所谓零个输入是指算法本身定出了初始条件。

(4) 有一个或多个输出。一个算法有一个或多个输出,以反映对输入数据加工后的结果。没有输出的算法是毫无意义的。

(5) 有效性。算法中执行的任何计算步骤都是可以被分解为基本的可执行的操作步骤,即每个计算步骤都可以在有限的时间内完成(也称之为有效性)。

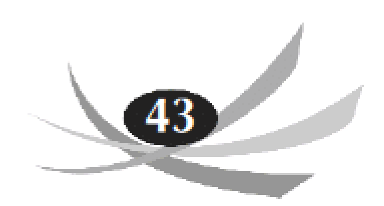
3.1.2 算法表示方法

为了表示一个算法,可以用不同的方法。常用的方法有:自然语言、传统流程图、结构化流程图、伪代码等。下面针对判定一个大于或等于3的正整数是否是素数的例子来介绍这些表示方法。

所谓素数,是指除了1和该数本身之外,不能被其他任何整数整除的数。判定一个大于或等于3的正整数n是否是素数的具体的操作方法是:将n作为被除数,将2到(n-1)的各个整数先后作为除数,如果都不能被整除,则n为素数。

1. 用自然语言表示算法

自然语言就是人们日常使用的语言,可以是汉语、英语或其他语言。对于判定素数的



例子，用自然语言描述的算法如下：

S1：输入一个大于或等于3的数n。

S2：i＝2(i作为除数)。

S3：n除以i，得余数r。

S4：如果r＝0，表示n能被i整除，则输出n“不是素数”，算法结束，否则执行S5。

S5：i＝i＋1。

S6：如果i≤n－1，返回S3，否则输出n是素数，算法结束。

实际上，n必须被2到(n－1)的整数除，只需被2到$\sqrt{n}$之间的整数除即可。因此，步骤S6可以改为：

S6：如果i≤$\sqrt{n}$，返回S3，否则输出n是素数，算法结束。

通过上面这个例子可以知道用自然语言表示算法时通俗易懂，但文字冗长，且容易出现歧义性。此外，用自然语言描述包含分支和循环的算法时不是很方便。因此，除了那些很简单的问题以外，一般不用自然语言描述算法。

2. 用流程图表示算法

流程图是用一些图框来表示各种操作。用流程图表示算法，直观形象，且易于理解。美国国家标准协会ANSI(American National Standard Institute)规定了一些常用的流程图符号(见图3-1)，这些流程图符号已作为一种标准被广泛采用。

图3-1中平行四边形(第二个)用于数据的输入或输出，菱形框的作用是对于一个给定的条件进行判断，根据条件的成立与否选择相应的操作。它有一个入口，两个出口，但最终只选择其中的一个出口，如图3-2所示。

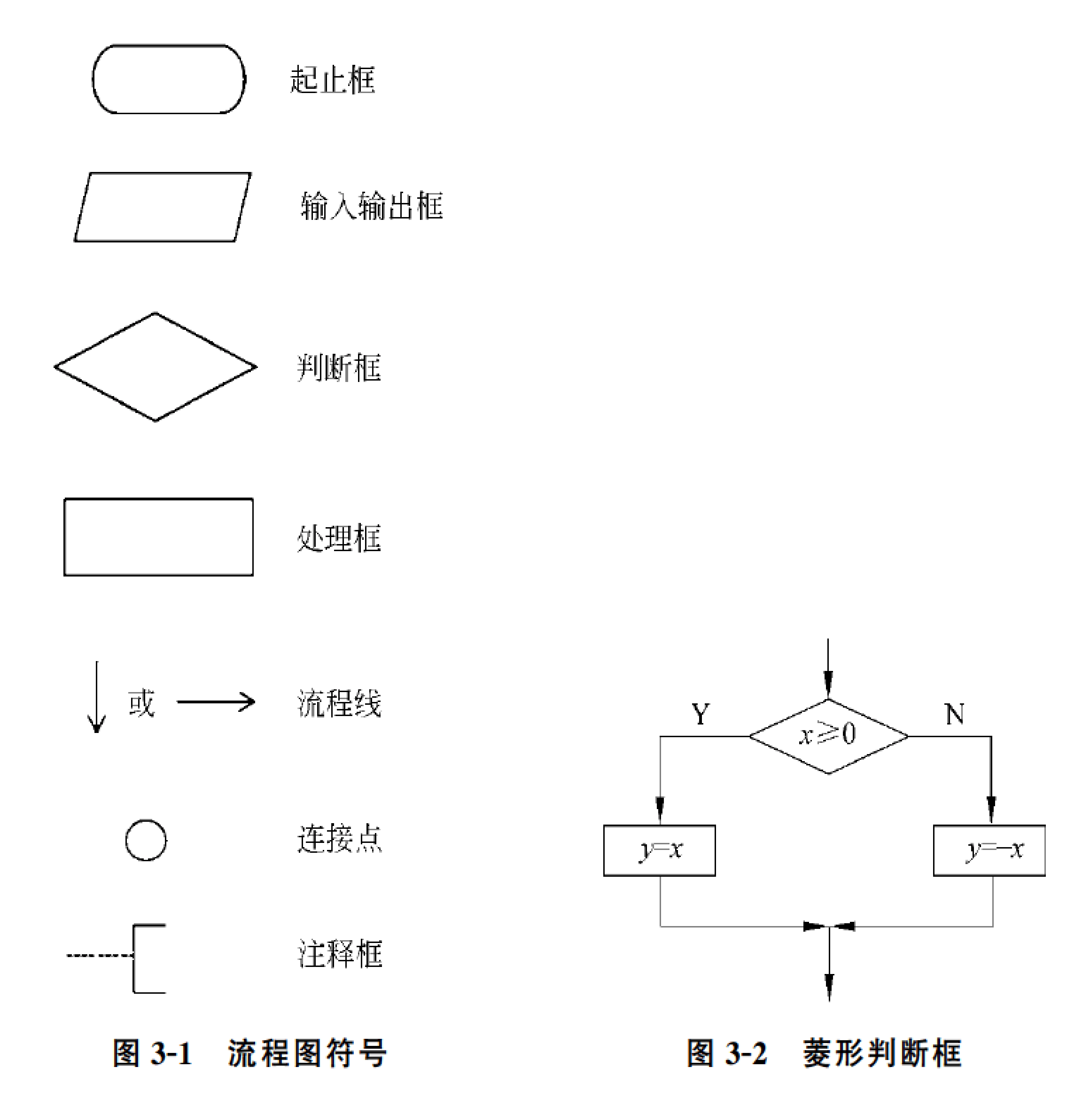


图3-1　流程图符号　　　图3-2　菱形判断框

矩形框主要用于计算赋值操作。小圆圈的作用是将画在不同地方的流程线连接起来。当使用一个流程图画不下而分开画时就需要连接点，这样可以避免使用一个流程图画而导致流程线的交叉或过长，从而使得流程图更加清晰。注释框不是流程图中必要的部分，只是为了对流程图中某些框的操作做必要的补充说明，使得阅读流程图的人更好地理解流程图的作用。

对于判定素数的例子，用流程图表示如图 3-3 所示。

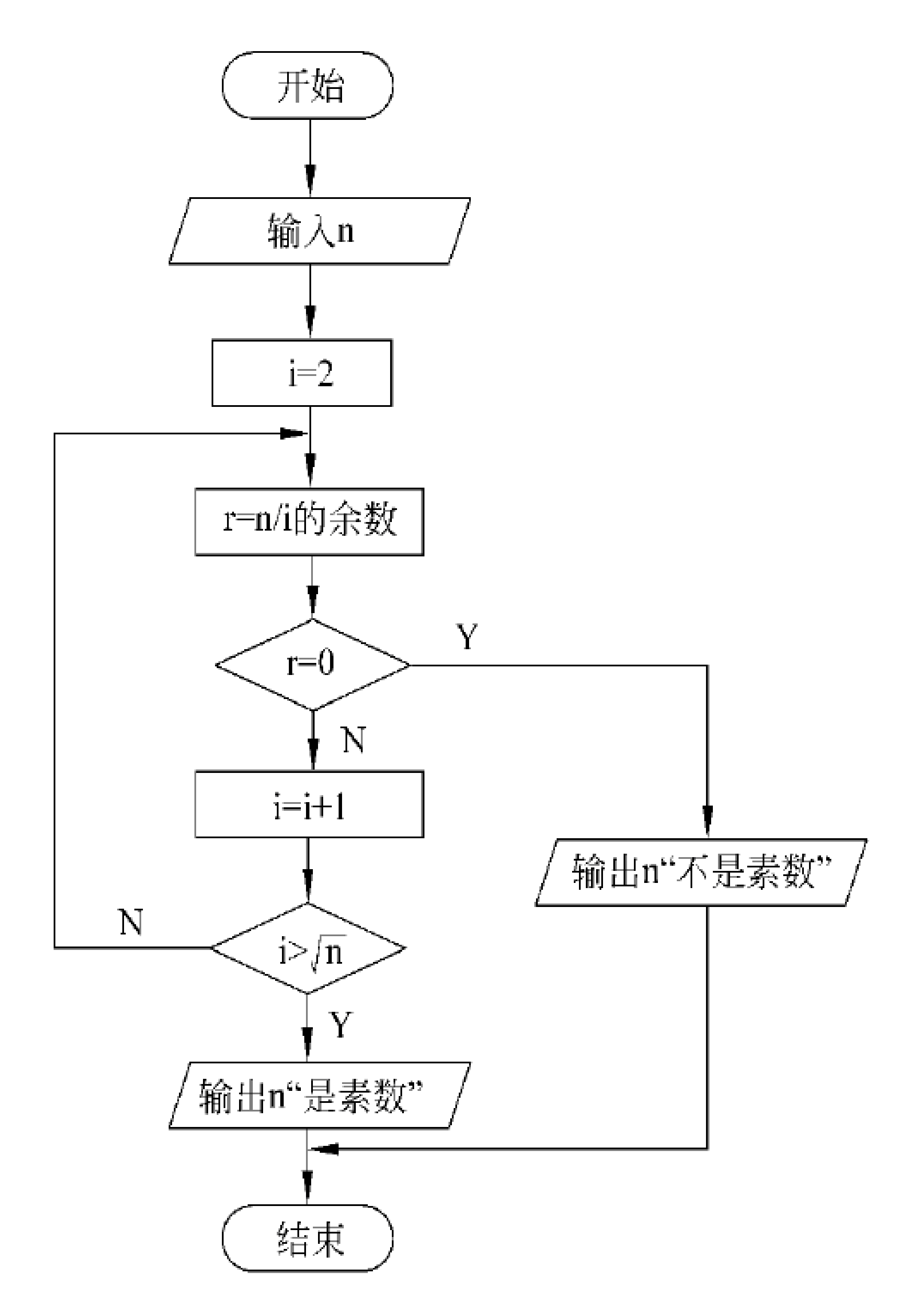


图 3-3 判定素数的流程图

3. 用 N-S 结构化流程图表示算法

1973 年美国学者 I. Nassi 和 B. Shneiderman 对传统流程图进行改进，提出了一种新的流程图形式。在这种流程图中完全去掉了带箭头的流程线。算法的所有操作都写在一个大的矩形框内，在这个矩形框内可以包含顺序结构、选择结构和循环结构这三种基本结构。这种流程图又称为 N-S 结构化流程图(N 和 S 是两位美国学者的英文姓氏的首字母)。

注意： 三种基本结构都具有以下几个共同的特点：

(1) 只有一个入口。

(2) 只有一个出口。

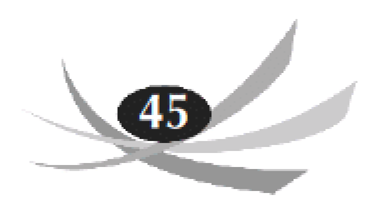
(3) 结构体内的每一部分都有机会被执行到。

(4) 结构体内不存在"死循环"。

N-S 流程图用以下流程图符号来表示算法。

(1) 顺序结构。顺序结构用图 3-4 所示，由 A 和 B 两个框组成。

(2) 选择结构。选择结构用图 3-5 所示，由条件 p、A 和 B 框组成，当条件 p 成立时执



行A操作，p不成立则执行B操作。

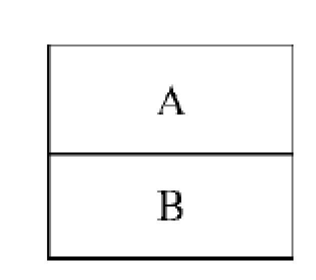


图3-4　顺序结构

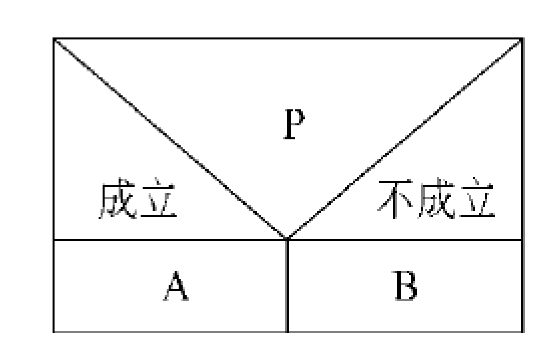


图3-5　选择结构

(3) 循环结构。循环结构又分成当型循环和直到型循环。当型循环结构用图3-6来表示。当条件p成立时重复执行A操作，直到条件p不成立为止。直到型循环用图3-7表示。注意：Python语言没有直到型循环，但在其他语言(例如C语言)中就有直到型循环。

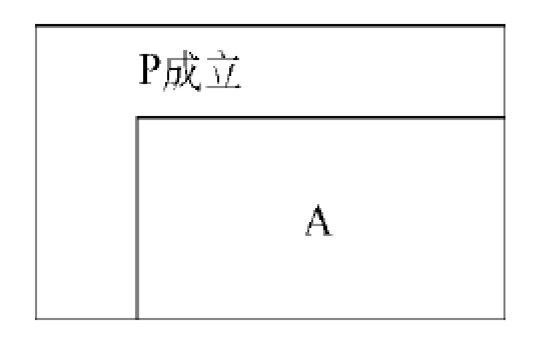


图3-6　当型循环结构

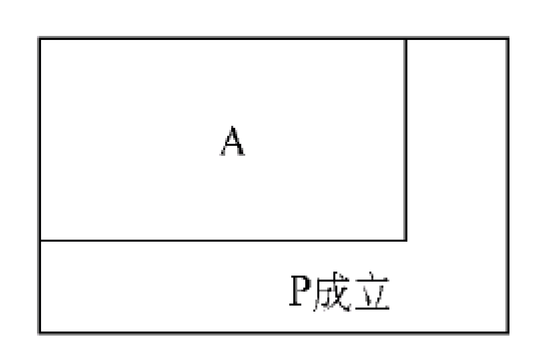


图3-7　直到型循环结构

注意：在图3-4～图3-7中的A框或B框，可以是一个简单的操作(如输入数据或输出数据等)，也可以是三种基本结构之一，即基本结构之间可以相互嵌套。使用以上三种N-S流程图中的基本框就可以组成复杂的N-S流程图。使用N-S结构化流程图表示算法时更加直观、形象，且易于理解。

对于判定素数的例子，从图3-3可看出，这个流程图不是由三种基本结构组成的。图中的循环部分有两个出口(一个是第二个判断框条件不成立时下面的出口，另一个是第一个判断框条件成立时右边的出口)，不符合基本结构的特点。要想用N-S流程图来表示，就必须要对图3-3做适当的修改，使其可以分解为三种基本结构。现在的问题点就是如何把图中循环部分的两个出口合并成一个出口。我们可以这样做：当r=0时，不直接输出n“不是素数”，而是设一个标识值(变量f，初始值为0，表示n是素数)。当r≠0时保持f=0，一旦r=0，则把f变成1，此时说明n不是素数。然后将f=1处理框的出口改为指向第二个判断框，同时将第二个判断框中的条件改为“$i\leqslant\sqrt{n}$和f=0”，即只有当“$i\leqslant\sqrt{n}$”和“f=0”这两个条件都成立时才继续执行循环，否则就跳出循环。接下来就可以通过判断标识值f是否为0，如果为1，则说明在循环当中，某一次n/i的余数r的值为0，使得f的值变成1。如果f的值仍为0，则表示在上面的每次循环中，n都不能被每一个i整除。因此可以得出，当f=1时，n是素数，输出n是素数的信息，当f≠1(即f=0)时，n不是素数，输出n不是素数的信息。修改后的流程图如图3-8所示。对于判定素数的例子，用N-S结构化流程图表示如图3-9所示。

注意：图3-9中用到了直到型循环，它的判断条件为“直到$i>\sqrt{n}$或f≠0”，即只有“$>\sqrt{n}$”或“f≠0”其中之一成立，就不再继续执行循环。

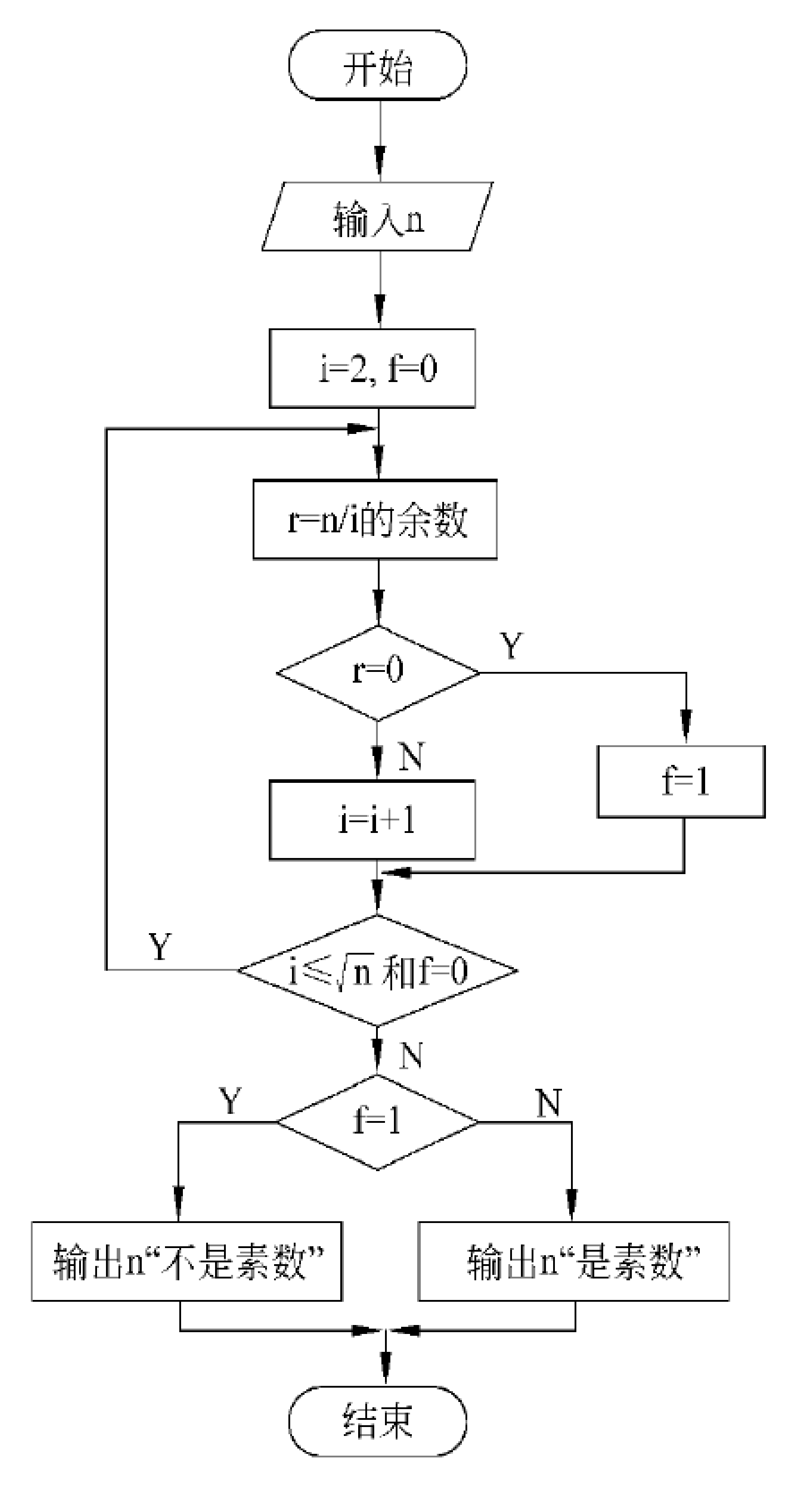



图 3-8 评定素数符合结构化的流程图

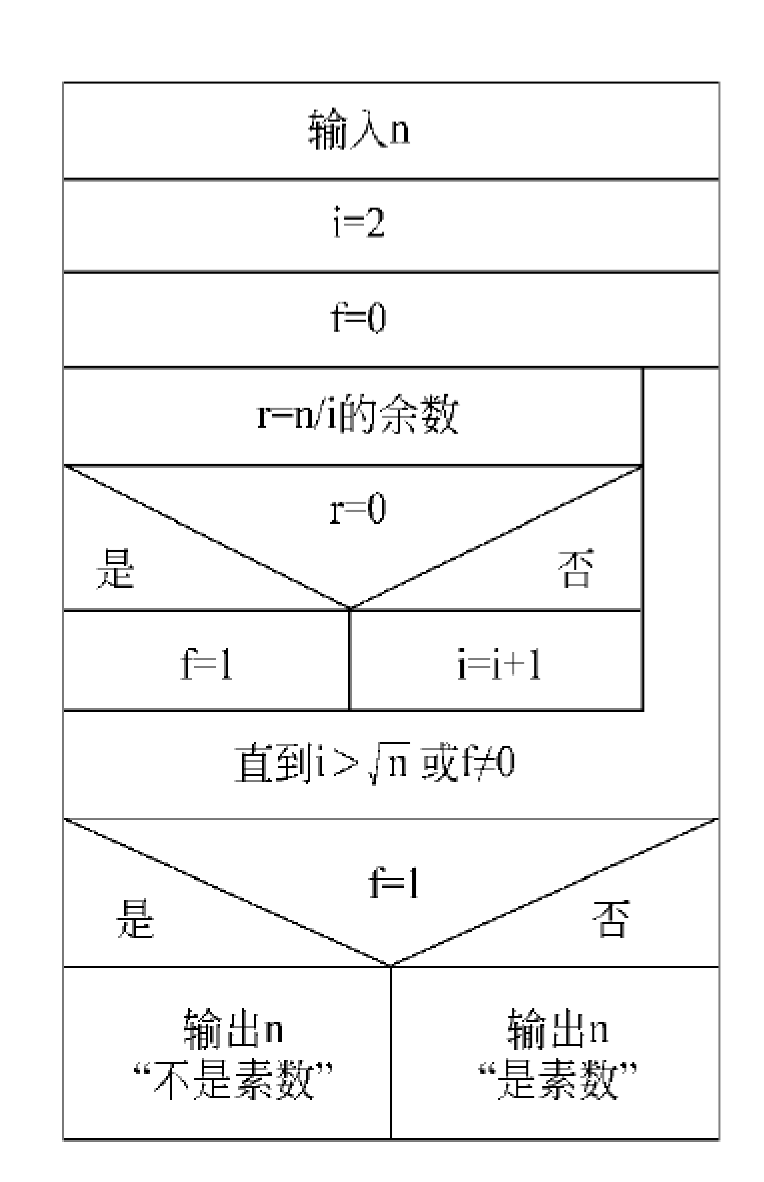


图 3-9 评定素数的 N-S 流程图

4. 用伪代码表示算法

用传统的流程图和 N-S 流程图表示算法虽然直观易懂,但画起来比较费事。在设计一个算法时,可能要反复修改,而修改流程图是比较麻烦的。因此,流程图适合于表示一个算法,但在设计算法的过程中,尤其是当算法比较复杂、可能需要反复修改时,使用流程图显然不是很理想。为了设计算法的方便,常常使用伪代码来描述算法。

伪代码是一种用介于自然语言和计算机语言之间的用文字和符号来描述算法的语言。使用伪代码的目的是使被描述的算法可以容易地以任何一种编程语言(Python、C、Java 等)实现,同时用伪代码来描述的算法也比较好理解。

用伪代码书写算法并无固定的、严格的语法规则,只要把意思表达清楚,并且书写的格式要尽量写得清晰易懂。

对于判定素数的例子,用伪代码表示算法如下:

```
begin                    #算法开始
    输入 n
    i=2
    f=0
    r=n%i
    while(i≤√n和 f=0)
    {
```

```
        if(r==0)
            f=1
        else
            i=i+1
    }
    if(f==1)
            输出 n"不是素数"
        else
            输出 n"是素数"
end                     #算法结束
```

5. 用计算机语言表示算法

当算法设计好之后,要想交给计算机来解题,就必须要将设计好的算法(用流程图或伪代码表示)转化为计算机语言,即只有用计算机语言编写的程序才能够被计算机执行。注意:用计算机语言表示算法必须要遵循所使用的语言的规则。

对于判定素数的例子,用计算机语言(Python)表示算法如下:

```
#coding:utf-8
#例 3-1  判定一个大于或等于 3 的正整数是否是素数
import math             #导入 math 包
n=input("输入一个大于或等于 3 的正整数")

#初始化阶段
i=2
f=0

#循环计算阶段
while(i<=math.sqrt(n) and f==0):
    r=n%i
    if(r==0):
        f=1
    else:
        i=i+1

#终止阶段
if(f==1):
    print "%d,%s" % (n,"不是素数")
else:
    print "%d,%s" % (n,"是素数")
```

注意:为了更好地掌握 Python 编程,从这章开始,例子中的代码(除第 4、第 5 章部分例子外)不再以交互式的模式下编辑和执行,而是以源文件的形式编辑和执行。

程序的第一行加上了#coding:utf-8,这一行的作用是声明文件的编码格式为 UTF-8,使得程序能够正常显示中文。UTF-8(8-bit Unicode Transformation Format)是一种

针对 Unicode 的可变长度字符编码，是目前流行的编码格式，可以显示中文简体、繁体及其他语言（如英文、日文、韩文）。

程序第二行的作用是把 math 包（该包提供了非常丰富的函数）导入当前程序执行环境。这样，程序就可以使用 math 包所提供的函数了，如执行 math. sqrt(n)这条语句就可以求出 n 的平方根。包的相关知识将在第 5 章中介绍。

程序中使用了 if 条件判断语句和 while 循环语句，这些内容将在下一节中介绍。

本小节介绍了算法的表示方法，那么如何才能够设计出结构化的算法呢？下面将介绍一种结构化的设计方法——自上而下求精法。

3.1.3 自上而下求精法

自上而下求精法就是先从总体上分析解决问题的大体步骤，然后对前一次求精结果再进行细化，得到这次的求精结果，如果该结果的求解过程已经很明确，那么就结束，否则，还要进一步求精，直至问题求解过程很明确。

下面要对求平均成绩问题进行“泛化”。开发一个求平均成绩的程序，该程序每次执行时，都能出来任意数量的成绩，在这个例子中，我们事先不假定要输入多少个成绩，程序必须能处理任意数量的成绩。那么，程序什么时候应该停止成绩输入呢？什么时候应该计算，什么时候则应打印平均成绩呢？

为解决这个问题，一个办法是使用一个特殊值作为标识来指出“输入结束”。用户要不断输入成绩，直到所有成绩都输入完毕，然后，可以输入一个作为标识的特殊值，指明已经输入了最后一个成绩。

显然，这个作为标识的特殊值是有讲究的，否则可能与正式输入产生混淆。由于测验成绩通常都是非负的正数，所以－1 对该问题来说可以作为标识的特殊值。因此，执行一遍这个求平均成绩的程序，可能得到像“80，90，85，92，88，－1”这样的一连串输入。遇到－1，程序应该能马上计算并打印出 80，90，85，92，88 这 5 个值的平均值。

为生成求平均成绩的程序，我们采用一种名为“自上而下求精法”的方法，它有助于开发具有良好结构的程序。首先用文字来表示“顶部”：

求解测验的平均成绩

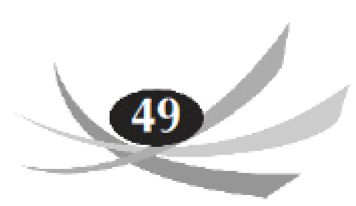
在这个顶部中，它概括了程序的总体功能。因此，最上部的语句实际上是对程序的一个完整的描述。但顶部无法传达足够的细节，无法据此写一个完整的 Python 程序。所以，接下来要进行“求精”。首先，将顶部的语句分解成一系列更小的任务，然后按它们的执行顺序排列好，这样便得到了“第一次求精”的结果：

初始化变量

输入、总计和计数测验成绩

计算并打印平均成绩

这里只使用了顺利结构，即上述步骤会按顺序执行。虽然可以根据“第一次求精”的结果来写出一个完整的 Python 程序，但思路可能还是不够清晰，而且对于一些输入可能不会正确算出结果。从总体来说，虽然能基本解决这个问题，但还是比较粗糙的，需要进行下一级（第二级求精），需要用到一些特定的变量，程序要维持一个不断变化的 total 值，



统计处理了多少个成绩的 count 值，包含了每个成绩值的一个变量，以及包含了计算好的平均成绩的一个变量。

　　初始化变量

可“求精”为：

　　将总和初始化成零

　　将计数器初始化成零

　　输入、总计和计数测验成绩这个步骤需要用到一个循环结构来实现，它能连续输入每个成绩。由于事先不知道要输入的成绩个数，所以计划用一个特殊值来控制程序的循环输入。用户可不断输入合法的成绩值。最后一个合法成绩输入完毕后，在下一次重复时，就输入这个特殊值来结束输入。程序会在每次输入成绩后检测是否为特殊值，如果是则终止循环。

　　输入、总计和计数测验成绩

可“求精”为：

```
    输入第一个成绩（可能是特殊值）
    只要用户还没输入特殊值
        把这个成绩加到总和里
        成绩计数器自增 1
    输入下一个成绩（可能是特殊值）
    计算并打印平均成绩
```

可“求精”为：

```
    假如计数器不等于 0
        将平均值设为总和除以计数器值
        打印平均值
    否则
        打印“没有输入成绩”
```

　　综上所述，求平均值问题的完整二次求精结果：

```
    将总和初始化成零
    将计数器初始化成零
    输入第一个成绩（可能是特殊值）
    只要用户还没输入特殊值
        把这个成绩加到总和里
        成绩计数器自增 1
    输入下一个成绩（可能是特殊值）
    假如计数器不等于 0
        将平均值设为总和除以计数器值
        打印平均值
    否则
        打印“没有输入成绩”
```

根据上述的算法过程就可以写出求平均成绩的完整的 Python 程序：

```
#coding:utf-8
#例 3-2  自上而下求精法
#初始化阶段
total=0
gradeCounter=0

#计算阶段
grade=raw_input("输入成绩,-1则结束输入:")
grade=float(grade)

while grade !=-1:
    total=total+grade
    gradeCounter=gradeCounter+1
    grade=raw_input("输入成绩,-1则结束输入:")
    grade=float(grade)

#终止阶段
if gradeCounter !=0:
    average=total/gradeCounter
    print "平均成绩为:",average
else:
    print "没有成绩输入。"
```

通过这个求解平均成绩的例子，相信读者已经基本掌握了使用“而上而下求精法”来求解问题的方法，接下来介绍控制结构。

3.2 控制结构

通常程序中的语句是按原先书写的顺序执行的，这称为“顺序执行”。然而，Python 允许程序员自行控制接着要执行的语句，这是由 Python 的“控制结构”来实现的。

20 世纪 60 年代，大量事实证明，滥用 goto 语句(在 BASIC 和 C 等语言中使用的程序控制语句)将使程序流程变得杂乱无章，没有规律性，且可读性差。直到 20 世纪 70 年代，人们想出了一种结构化程序设计方法。使用这种方法编写出来的程序条理更清晰、更易于调试和修改，使得一个软件项目的开发时间、速度、成本等方面都得到了很大的改善。因而，这种结构化程序设计方法受到了广大程序设计者的青睐，这种方法主张尽量不用或少用 goto 语句。

研究表明，只需要三种程序控制结构便可写出所有的程序。这三种控制结构包括：顺序结构、选择结构以及循环结构。其中，顺序结构是所有计算机语言默认的结构。除非特别声明，否则计算机都会从上往下顺序执行。

Python 语言提供三种类型的选择结构，它们分别通过 if、if/else 和 if/elif/else 这三种语句来实现。对于 if 选择结构，如果给定的条件成立(true)，则会执行一条或一段语

句;否则跳过这条或这段语句。对于if/else选择结构,如果条件成立(true),则会执行一条或一段语句,否则,执行另一条或另一段语句。如果是if/elif/else选择结构,则根据条件的成立与否在多个不同的语句中选择一个执行。这三种类型的选择结构将在下一节中分别介绍。

Python语言提供两种类型的循环结构,它们分别通过while和for语句来实现。while语句是当条件成立(true)时,执行while语句中的内嵌语句,其特点是先判断条件,后执行语句。for语句可以循环遍历任何序列的数据对象,如一个列表或者一个字符串等。

接下来分别介绍选择结构和循环结构。

3.3　选择结构

选择结构是三种基本结构之一,它的作用是根据所指定的条件是否满足,而执行相对应的操作。Python提供了三种类型的选择结构:if、if/else和if/elif/else,这三种类型的选择结构各有各的特点,下面分别讨论这三种选择结构。

3.3.1　if选择结构

if选择结构是通过一条关系表达式的执行结果(true或者false)来决定执行的代码块。可以通过图3-10来理解if选择结构的执行过程。

Python程序语言指定任何非零和非空(null)表达式的值为true;零或者空为false。

Python编程中if语句用于控制程序的执行,基本形式为:

```
if 判断条件:
    执行语句……
```

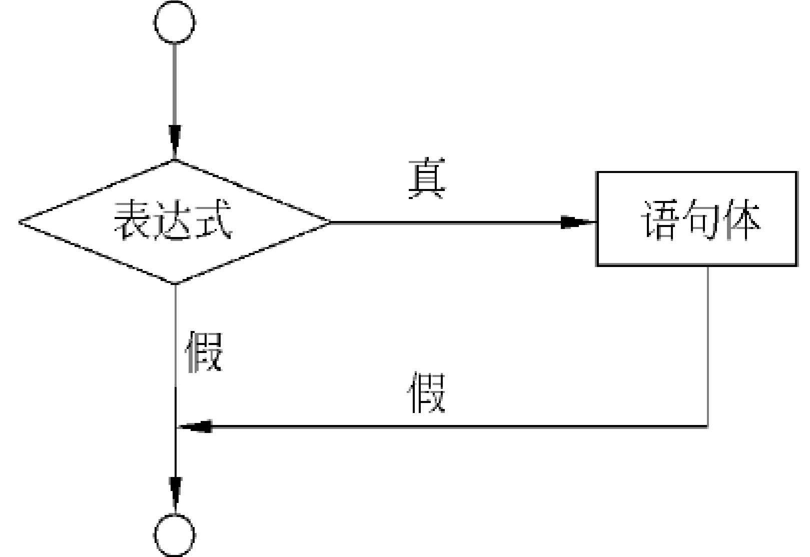


图3-10　if选择结构流程图

其中"判断条件"成立时(非零),则执行后面的语句,执行语句可以是单个语句或语句块(多条语句组成)。

注意:判断条件后面的冒号必须要有。对于语句块,Python利用缩进量是否一致来表示是否属于同一个语句块,其他程序语言通常使用大括号来表示同一个语句块。Python对缩进的要求非常严格,同一个语句块中的每一条语句的缩进量必须保持一致,否则程序会无法运行。初学者往往混用空格键个Tab键来缩进,使得同一个语句块中的语句缩进量不一致而导致出现无法执行。所以,建议都采用一个Tab键来对同一个语句块中的语句进行缩进。

下面通过例3-3和例3-4来理解if选择结构。

例3-3　设置一个标志变量flag,初始值为false。输入一个字符串,赋给变量name,然后判断该变量的值是否为"python",如果是则将标志变量flag的值置为true,并输出"welcome boss"信息。

程序如下：

```
#coding:utf-8
#例 3-3  if选择结构基本用法
flag=False
name=raw_input("输入变量 name 的值")
if name=='python':                #判断变量否为'python'
    flag=true                     #条件成立时设置标识为真
    print 'welcome boss'          #输出欢迎信息
```

如果输入的值是“python”，则输出结果为：

```
welcome boss
```

该程序对应的流程图如图 3-11 所示。

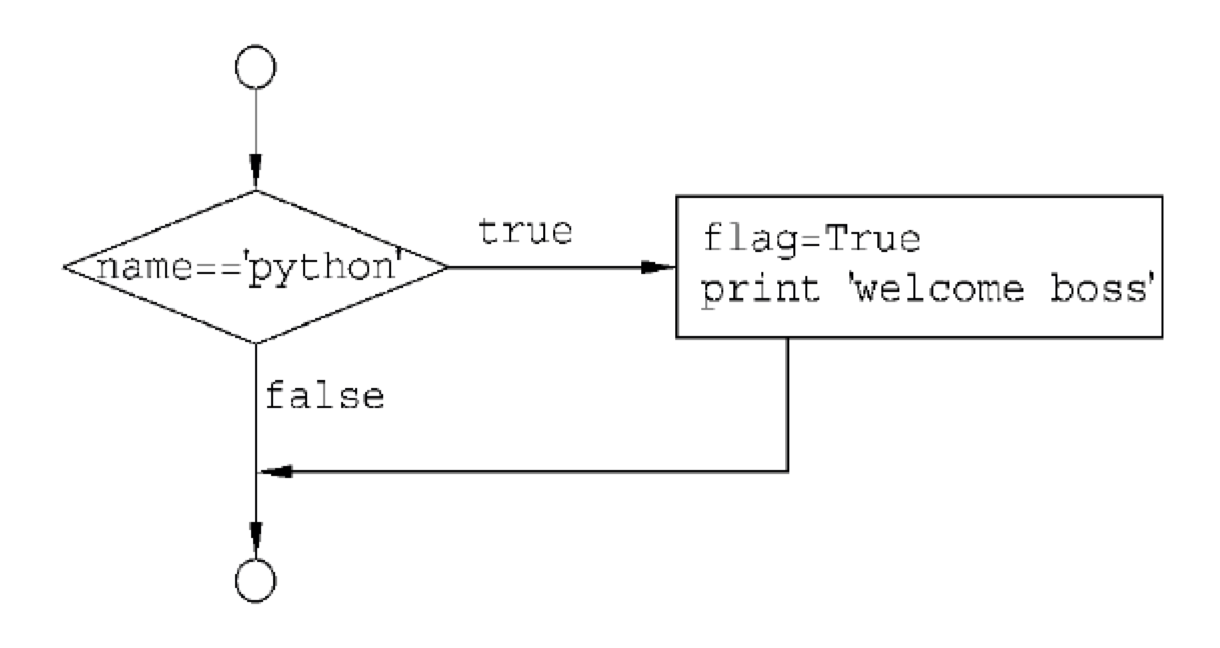


图 3-11 例 3-3 的 if 选择结构流程图

例 3-4 输入两个数，以从小到大的顺序输出这两个数。

这个问题的算法很简单，比较输入的变量 a 和 b，如果 a>b，则交换 a 和 b 的值。最终，a 的值都不超过 b 的值，按顺序输出 a、b 的值即可。

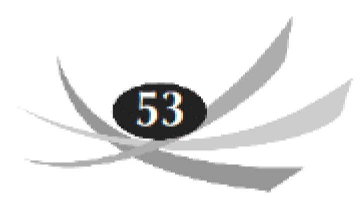
程序如下：

```
#coding:utf-8
#例 3-4  if选择结构基本用法
a=input("输入变量 a 的值")
b=input("输入变量 b 的值")
if a>b:
    t=a                           #t是用来交换变量 a、b 的中间变量
    a=b
    b=t
print "%d,%d" % (a,b)             #顺序输出变量 a、b 的值
```

if 语句的判断条件可以是关系表达式或者是由逻辑运算符连接关系表达式或逻辑量而组成的逻辑表达式。

3.3.2 if/else 选择结构

对于 if 选择结构，如果条件为 true，程序会执行指定的动作，否则会跳过该动作。而



对于 if/else 选择结构，程序员可针对条件成立与不成立这两种情况，分别指定一项动作。可以通过图 3-12 来理解 if/else 选择结构的执行过程。

if/else 选择的基本形式为：

```
if 判断条件:
    执行语句……
else:
    执行语句……
```

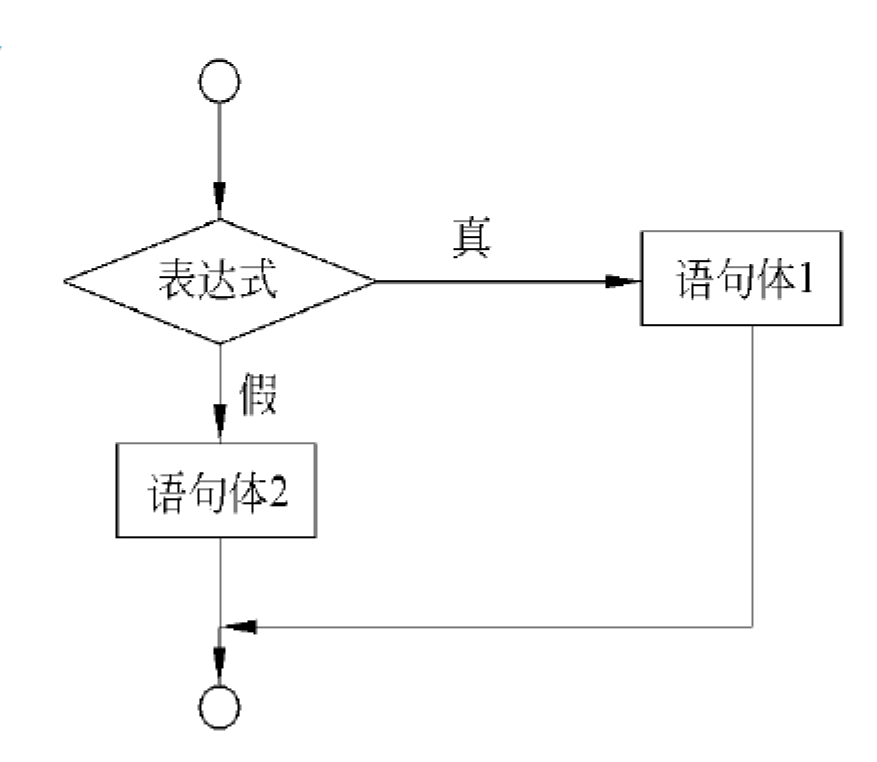


图 3-12　if/else 选择结构流程图

与前面的 if 选择结构一样，只是这里多了 else 字句，它表示当需要在条件不成立时执行后面的语句。

前面的例子加上 else 字句后的程序如下：

```
#coding:utf-8
#例 3-5  if/else 选择结构基本用法
flag=False
name=raw_input("输入变量 name 的值")
if name=='python':              #判断变量否为'python'
    flag=true                   #条件成立时设置标识为真
    print 'welcome boss'        #并输出欢迎信息
else:
    print name                  #条件不成立时输出变量的值
```

如果输入的值是“java”(无须输入双引号)，则输出结果为：

```
java
```

该程序对应的流程图如图 3-13 所示。

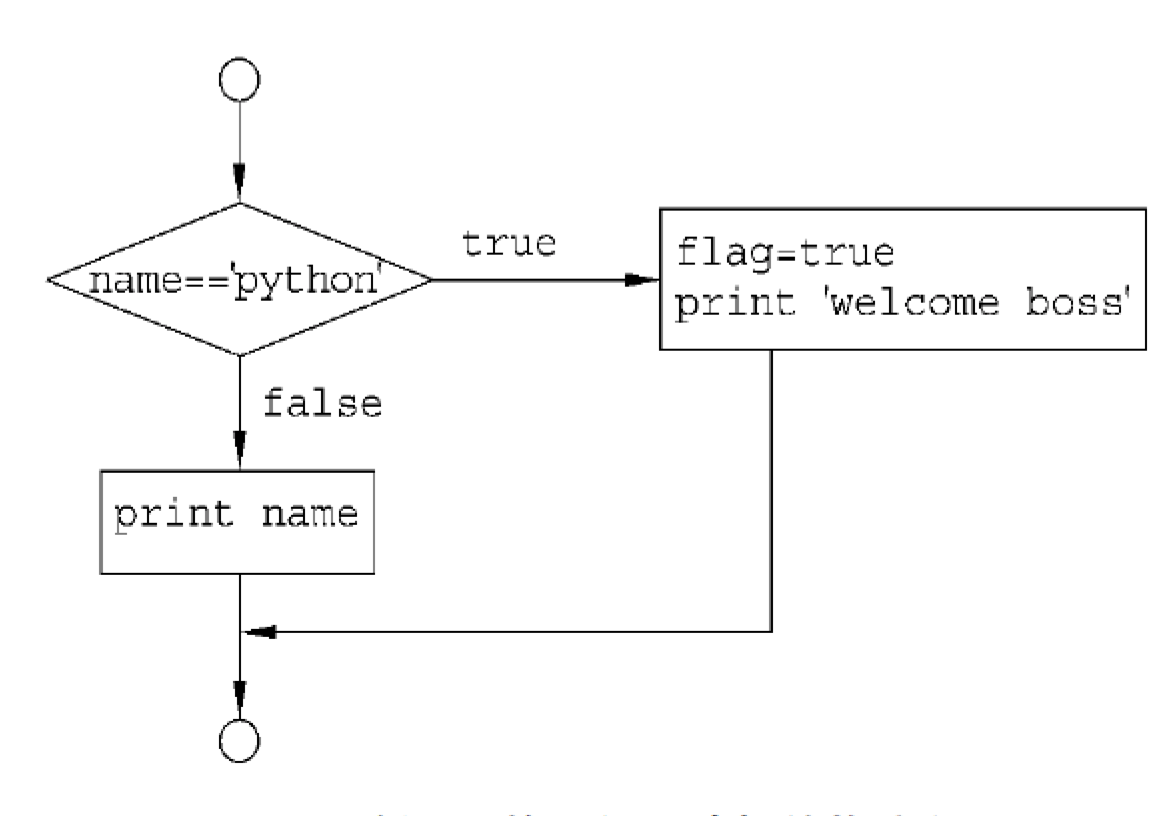


图 3-13　例 3-5 的 if/else 选择结构流程图

当判断条件为多个值时，可以用嵌套的 if/else 选择结构。它的做法是将一个 if/else 选择结构放入另一个 if/else 选择结构中。

例 3-6 假如学生成绩大于或者等于 90 分则打印 A，80～90 分之间打印 B，70～79 分之间打印 C，60～69 分之间打印 D，小于 60 分打印 E。

程序如下：

```
#coding:utf-8
#例 3-6 嵌套 if/else 选择结构用法
grade=input("输入成绩 grade")
if grade>=90:
    print "A"
else:
    if grade>=80:
        print "B"
    else:
        if grade>=70:
            print "C"
        else:
            if grade>=60:
                print "D"
            else:
                print "E"
```

注意：该程序有多层嵌套，编写程序的时候要小心，确保同一个语句块中的语句的缩进量保持一致。

如 grade 大于或等于 80，会进入最外层 if/else 结构的 else 语句中，符合这个语句中的条件，则会打印出 B。该程序对应的程序流程图如图 3-14 所示。

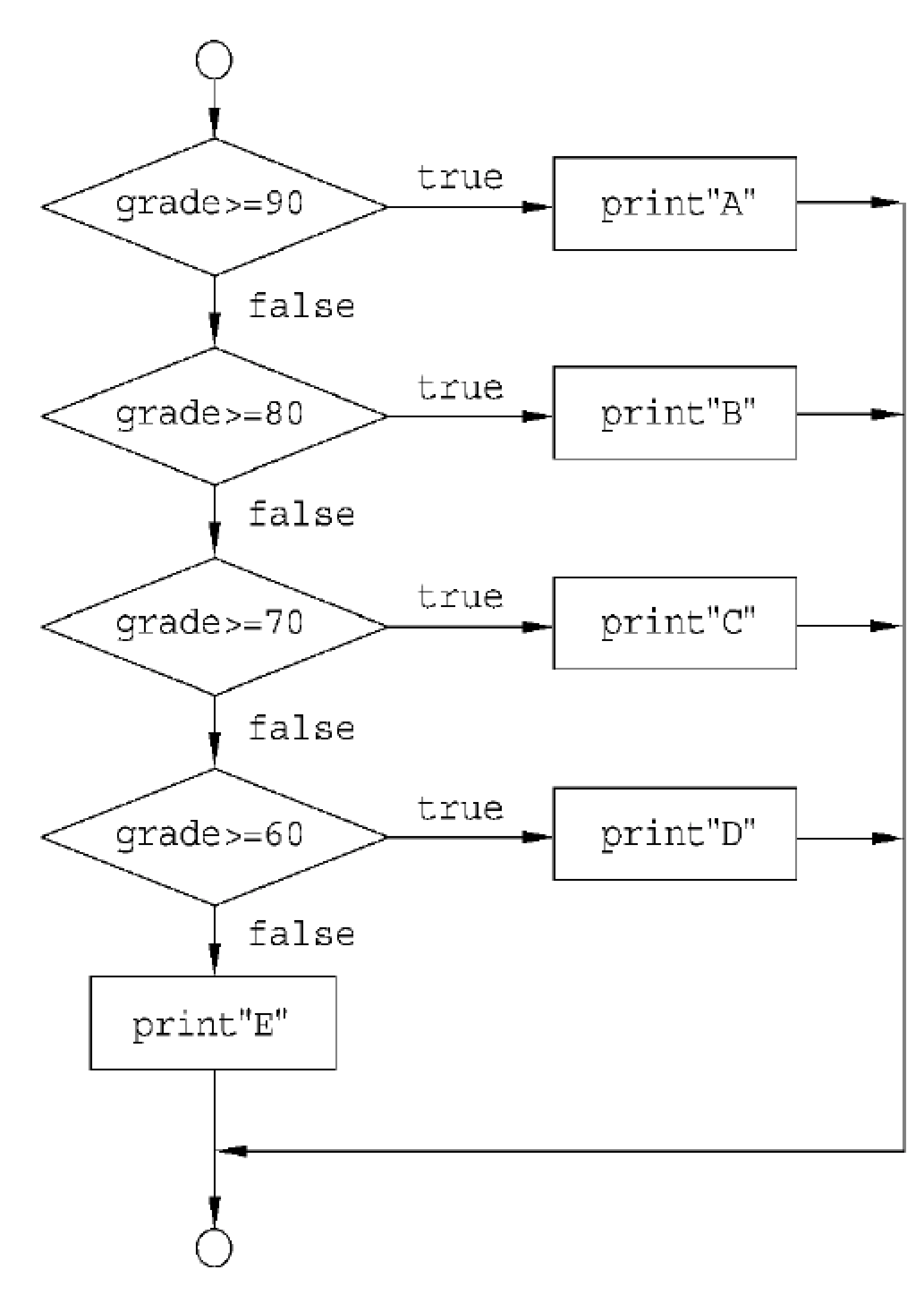
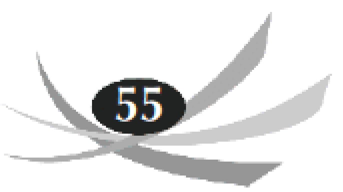


图 3-14 例 3-6 的嵌套 if/else 选择结构流程图



对于嵌套的 if/else 选择结构，Python 程序员更喜欢使用 if/elif/else 选择结构。下面介绍 if/elif/else 选择结构。

3.3.3　if/elif/else 选择结构

if/elif/else 结构可以用嵌套的 if/else 结构来替换，这两种形式效果是一样的，但 Python 程序员更喜欢后者，因为其避免代码过分向右边缩进，同时其可读性比前者要高。

可以通过图 3-15 来理解 if/elif/else 选择结构的执行过程。

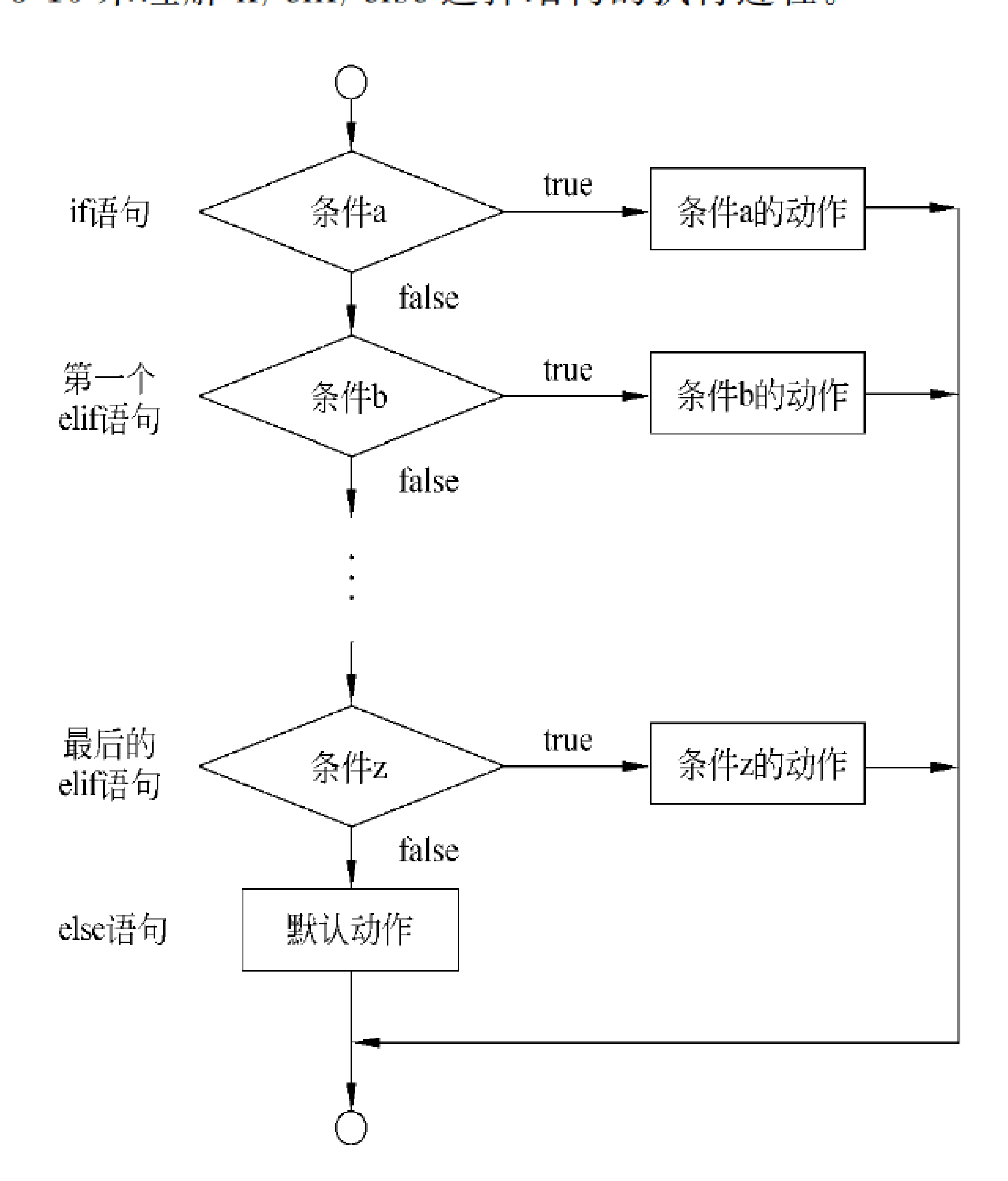


图 3-15　if/elif/else 选择结构流程图

if/elif/else 结构的形式：

```
if 判断条件 1:
    执行语句 1……
elif 判断条件 2:
    执行语句 2……
    .
    .
elif 判断条件 n:
    执行语句 n……
else:
    执行语句 n+1……
```

上述判断学生成绩所属等级的嵌套 if/else 结构转换为 if/elif/else 结构的 Python 程序如下：

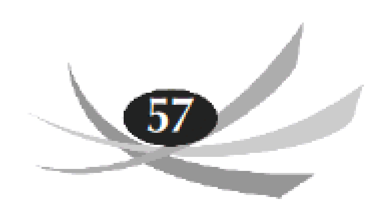
```
#coding:utf-8
#例 3-7  if/elif/else结构的用法
grade=input("请输入成绩")
if grade>=90:
    print "A"
elif grade>=80:
    print "B"
elif grade>=70:
    print "C"
elif grade>=60:
    print "D"
else:
    print "E"
```

由于 Python 并不支持 switch 语句，所以多个条件判断，只能用 if/elif/else 结构或嵌套 if/else 结构来实现。如果需要同时判断多个条件时可以使用第 2 章介绍的逻辑运算符。使用 or (或)，表示多个条件中只要有一个成立，则判断条件成功；使用 and (与)时，表示只有多个条件同时成立的情况下，判断条件才成功。

```
#coding:utf-8
#例 3-8  if语句多个条件
#使用 and
num=input("请输入一个数字")
if num>=0 and num<=10:          #判断值是否在 0~10之间
    print '%d is between 0 and 10' %num
else:
    print '%d is not between 0 and 10' %num

#使用 or
num=input("请输入一个数字")
if num<0 or num>10:             #判断值是否小于 0或大于 10
    print '%d is smaller than 0 or larger than 10' %num
else:
    print '%d is between 0 and 10' %num

#使用 and和 or
num=input("请输入一个数字")
#判断值是否在 0~5或者 10~15之间
if (num>=0 and num<=5) or (num>=10 and num<=15):
    print '%d is between 0 and 5 or between 10 and 15' %num
else:
    print '%d is not between 0 and 5 or not between 10 and 15' %num
```



如果依次输入的数字为15、5、8，则依次输出如下信息：

```
15 is not between 0 and 10
5 is between 0 and 10
8 is not between 0 and 5 or not between 10 and 15
```

当if有多个条件时可使用括号来区分判断的先后顺序，括号中的判断优先执行，此外还要注意第2章提到的逻辑运算符优先级低于关系运算符。

下面再举一个稍微复杂的例子。

例3-9　编程求方程$ax^2+bx+c=0$的根。

对于这个方程，有以下几种可能：

(1) $a=0$，是一个一次方程，根为$-c/b(b\neq 0)$。

(2) $b^2-4ac=0$，有两个相等的实根。

(3) $b^2-4ac>0$，有两个不等的实根。

(4) $b^2-4ac<0$，有两个共轭复根。

```
#coding:utf-8
#例3-9  求方程ax²+bx+c=0的根
import math                                                    #带入math包
a=input("input a")
b=input("input b")
c=input("input c")

if(a==0):                                                      #a=0,一次方程,根为-c/b(b≠0)
    print "The equation has one root:",-c/float(b)
else:
    disc=b*b-4*a*c
    if(math.fabs(disc)<=1e-6):                                 #b²-4ac=0,有两个相等的实根
        print "The equation has two equal roots:%f" %(-b/(2.0*a))
    else:
        if(disc>1e-6):                                         #b²-4ac>0,有两个不等的实根
            x1=(-b+math.sqrt(disc))/(2.0*a)
            x2=(-b-math.sqrt(disc))/(2.0*a)
            print "The equation has two distinct real roots:%f,%f" %(x1,x2)
        else:                                                  #b²-4ac<0,有两个共轭复根
            realpart=-b/(2.0*a)
            imagpart=math.sqrt(-disc)/(2.0*a)
            print "The equation has complex roots:"
            print "%f+%fi" %(realpart,imagpart)
            print "%f-%fi" %(realpart,imagpart)
```

程序中用disc代表b^2-4ac，先计算disc，以减少以后的重复计算。对于判断b^2-4ac是否等于0时，要注意：由于disc(即b^2-4ac)有可能为实数(如输入的值是小数时)，而

实数在计算和存储时会有一些微小的误差，因此不能直接判断“disc==0”，因为这样可能会出现本来是零的量，由于上述误差而被判别为不等于零而导致结果错误。所以采取的办法是判别 disc 的绝对值(math.fabs(disc))是否小于一个很小的数(例如这里设为 10^{-6})，如果小于此数，就认为 disc 等于 0。为了增加程序的可读性，用变量 realpart 代表复数的实部，变量 imagpart 代表复数的虚部。此外，当 a=0，求一次方程的根时，要把 b(或 a)转成实数类型，否则在进行相除运算时有可能由于除数和被除数都是整数而得到不正确的结果。如 a=0，b=3，c=5 时，如不进行上述转换，就会得到错误的结果−2，而正确的答案应该约为−1.666667。类似的，在本来除以(2 * a)的表达式中都把(2 * a)转换成(2.0 * a)，这样就可以得到准确的结果。

输入四组数据：

(1) 0、3、5

(2) 1、2、1

(3) 3、5、7

(4) 2、5、3

依次得到的输出为：

```
The equation has one root: -1.66666666667
The equation has two equal roots:-1.000000
The equation has complex roots:
-0.833333+1.280191i
-0.833333-1.280191i
The equation has two distinct real roots:-1.000000,-1.500000
```

注意：在嵌套 if/else 结构中，由于只要有一个条件满足，其余的判断语句就会终止执行，这比使用一系列的 if 结构执行速度要快一些。此外，把最可能成立的条件放在该嵌套 if/else 结构或 if/elif/else 结构的最前面，最不可能成立的条件放在最后面，这样可以有效减少判断条件的执行次数，使得嵌套的 if/else 结构或 if/elif/else 结构执行更快。

3.4 循环结构

在许多问题中都需要用到循环结构。例如，求某个班级的平均分；判断一个数是否是素数；求一个正整数的阶乘等。绝大多数的应用程序都包含循环。循环结构是结构化程序设计的基本结构之一，它和顺序结构、选择结构共同作为各种复杂程序的基本组成单位。Python 提供了两种类型的循环结构：while 循环和 for 循环。下面分别介绍 while 和 for 这两种循环结构。

3.4.1 while 循环结构

while 语句用于循环执行某段程序，即当给定的判断条件成立时，循环执行某段程序，以处理需要重复执行的相同任务。其基本形式为：

```
while 判断条件:
```

执行语句……

判断条件可以是任何表达式，任何非零、或非空(null)的值均为 true。当判断条件为 false 时，循环结束。可以通过图 3-16 来理解 while 循环的执行流程。

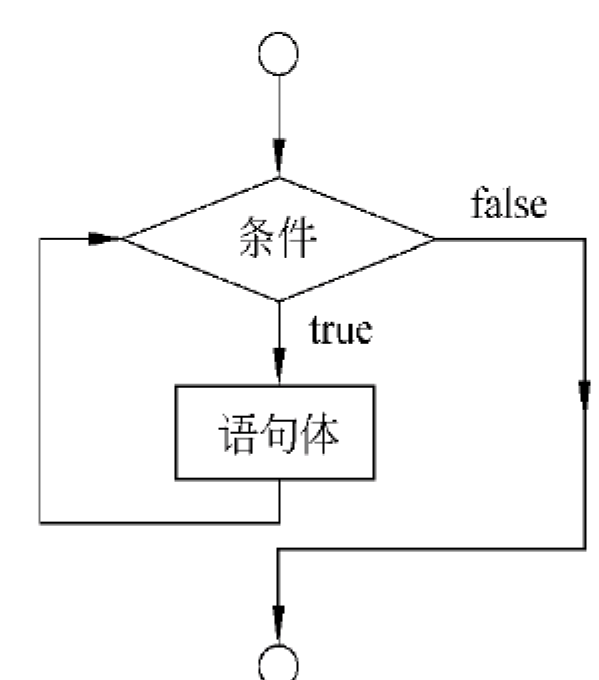


图 3-16　while 循环的执行流程图

下面通过例 3-10 和例 3-11 来掌握 while 循环。

例 3-10　求 $S_{100}=1+2+3+\cdots+100$。

程序如下：

```
#coding:utf-8
#例 3-10  while 循环的使用 1
#初始化
sn=0
an=1
while(an<=100):
    sn=sn+an
    an=an+1
print "The total of the S is %d",sn
```

输出结果：

```
The total of the S is 5050
```

注意：在循环体内要有使循环趋于结束的语句。例如本例中的"an=an+1"，若没有类似的使循环趋于结束的语句，则该循环会一直执行，永不结束。在编写含有循环结构的程序时特别要注意。

例 3-11　输入两个正整数 m 和 n，求其最大公约数和最小公倍数。

这题可以使用辗转相除法求最大公约数，而最小公倍数就等于两个数的乘积再除以最大公约数。

辗转相除法算法描述：当 $m \geq n$ 时(若 $m < n$，交换 m，n 的值)，m 对 n 求余为 r，若 r 不等于 0，则将 n 赋给 m，r 赋给 n，继续求余，直到 r 等于 0，此时，n 就为最大公约数。

程序如下：

```
#coding:utf-8
#例 3-11  while 循环的使用 2
m=input("input num m(m>0)")
while(m<=0):                              #若输入的 m≤0,则循环输入,直到 m>0
    m=input("input num m(m>0)")
n=input("input num n(n>0)")
while(n<=0):                              #若输入的 n≤0,则循环输入,直到 n>0
    n=input("input num n(n>0)")
s=m*n                                     #s 用来保存 m、n 的乘积
```

```
    if(m<n):                                #若m<n,则交换m、n的值
        t=m
        m=n
        n=t

    while(n!=0):                            #循环判断m/n的余数是否为0
        r=m%n
        m=n
        n=r

    gcd=m                                   #由于n赋给了m,所以m就是最大公约数
    lcm=s/gcd                               #s/gcd即为最小公倍数
    print "The GCD is%d,while the LCM is%d" % (gcd,lcm)
```

在循环结构中有两个重要的语句用来控制循环结构程序的执行。这两个语句分别是continue语句和break语句。continue语句用于跳过该次循环,而break语句则用于退出循环。此外,“判断条件”还可以是个常值,表示循环必定成立,这样就需要通过break语句来使程序跳出循环。

注意:如果条件判断语句永远为true,而循环体中又没有break语句使循环结束,循环将会无限地执行下去。在写循环程序时一定要小心,不能出现死循环。

下面通过例3-12来理解continue语句和break语句的用法。

```
#coding:utf-8
#例3-12  continue和break的用法
i=1
while i<=10:
    i=i+1
    if i%2==1:                              #奇数时跳过输出
        continue
    print i                                 #输出偶数2、4、6、8、10

i=1
while 1:                                    #循环条件为1必定成立
    print i                                 #输出1~10
    i=i+1
    if i>10:                                #当i大于10时跳出循环
        break
```

在Python中,while循环结构也可以和else语句一起使用。其中,while中的语句和普通的没有区别,else中的语句会在循环正常执行完(即while不是通过break来跳出循环的)的情况下执行。前面介绍的例3-1,判断一个数是否是素数。通过while语句和else语句结合可以使程序更简洁。

```
#coding:utf-8
#例 3-13  while 循环结构和 else 语句结合起来使用
import math                                          #导入 math 包
n=input("输入一个大于或等于 3 的正整数")

i=2
while(i<=math.sqrt(n)):
    r=n%i
    if(r==0):
        print "%d,%s" % (n,"不是素数")
        break
    else:
        i=i+1
else:
    print "%d,%s" % (n,"是素数")
```

该程序无须设置标记变量(标记某个数是否是素数),因为当循环体内余数 r 的值为零时,就说明该数不是素数,直接输出 n“不是素数”,然后跳出循环。注意,此时,由于是通过 break 语句而不是循环正常执行完使得程序结束,所以不会执行 else 的子句。当余数 r 一直都不为零,直到条件“i<=math.sqrt(n)”不满足时,跳出循环。此时,循环正常执行完,然后执行 else 子句,输出 n“是素数”。

其实,本例中的 else 子句相当于条件为“i>math.sqrt(n)”的 if 语句。

3.4.2　for 循环结构

通过上面的介绍,我们了解到 while 语句很灵活,它可以用在任何条件为真的情况下循环执行某段语句块。但如果需要遍历一个列表或集合中的元素,使用 while 语句就有些费事了。使用 for 语句就可以轻而易举地解决这个问题。

for 循环可以遍历任何序列的数据对象,如一个列表或者一个字符串。for 循环的基本形式如下:

```
for 变量(v) in 序列(q):
    语句(s)
```

其中,序列是指一系列元素的集合。循环第一次时,序列 q 中的第一项会指派给变量 v,并执行语句(s),以后每次循环时,都先将序列 q 中的下一项指派给变量 v,再执行语句(s)。当序列 q 中的每一项都执行了一次后,循环会终止。大多数情况下,for 结构可用等价的 while 结构表示。可以通过图 3-17 来理解 for 循环的执行流程。

下面通过例 3-14 来掌握 for 循环的用法。

```
#coding:utf-8
#例 3-14  for 循环结构的用法
print "输出'Python'中的每个字母:"
```

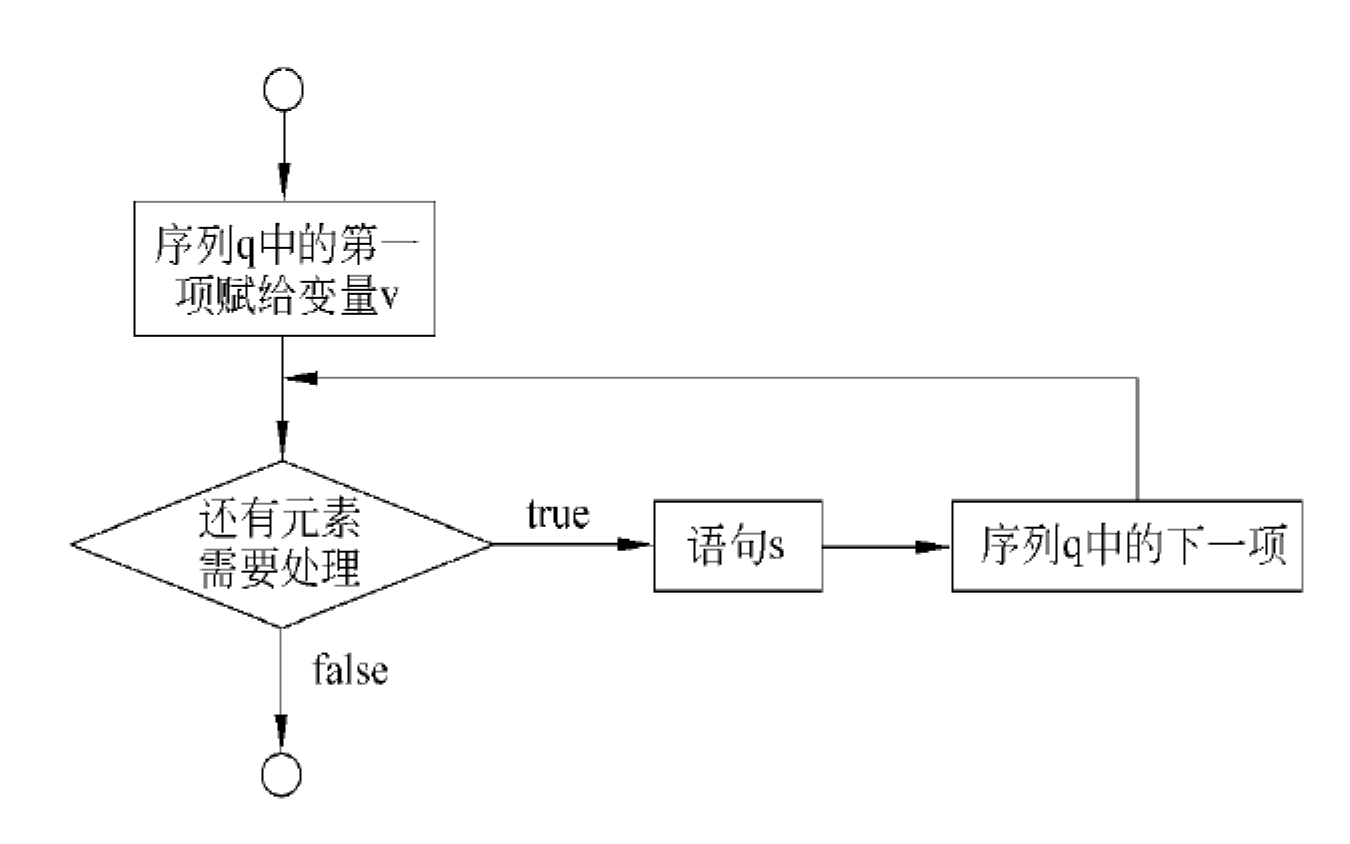


图 3-17　for 循环的执行流程图

```
for letter in 'Python':                         #序列为字符串"Python"
    print '当前字母:', letter

print "输出 1~4 的每个数字:"
for counter in range(1,5):                      #序列为 1~5(不包括 5)的数字列表
    print '当前数字:', counter

fruits=['banana', 'apple',  'mango']
print "常规方法遍历序列 banana','apple','mango':"
for fruit in fruits:                            #序列为有 3 个元素的列表
    print '当前水果:', fruit

fruits=['banana', 'apple',  'mango']
print "通过索引的方式遍历序列 banana','apple','mango':"
for index in range(len(fruits)):                #序列为 0~3(不包括 3)的数字列表
    print '当前水果:', fruits[index]
```

输出的结果如下所示：

```
输出'Python'中的每个字母:
当前字母: P
当前字母: y
当前字母: t
当前字母: h
当前字母: o
当前字母: n
输出 1~4 的每个数字:
当前数字: 1
当前数字: 2
当前数字: 3
当前数字: 4
常规方法遍历序列 banana','apple','mango':
```

```
当前水果: banana
当前水果: apple
当前水果: mango
通过索引的方式遍历序列 banana','apple','mango':
当前水果: banana
当前水果: apple
当前水果: mango
```

在本例中使用了 Python 内置函数 len()和 range()。其中,函数 len()返回列表的长度,即元素的个数。range 返回一个数字序列的列表,如 range(10)返回 0～10(不包含 10)的列表;range(5,10)返回 5～10(不包含 10)的列表;range(5,10,2)返回列表[5,7,9]等。

在 Python 中,for 循环结构和 while 结构一样,也可以和 else 语句一起使用。

一个循环体内又包含一个完整的循环结构,称为循环的嵌套。内层的循环中还可以嵌套循环,这就是多层循环。while 循环和 for 循环可以相互嵌套。

下面通过两个例子来掌握循环嵌套的用法。

例 3-15 输出九九乘法表,即第一行输出“1 * 1=1”,第二行输出“1 * 2=2 2 * 2=4”,以此类推。

程序如下:

```
#coding:utf-8
#例 3-15 输出九九乘法表
str=""                                      #用来保存要输出的字符串
for i in range(1,10):                       #i 的范围 [1~9]
    for j in range(1,i+1):                  #j 的范围 [1~i]
        #将每个乘法表达式拼接起来
        str=str+"%d*%d=%-2d "%(j,i,i*j)
    str=str+"\n"                            #内层循环结束后再拼接一个换行符
print str
```

输出结果如下所示:

```
1*1=1
1*2=2  2*2=4
1*3=3  2*3=6   3*3=9
1*4=4  2*4=8   3*4=12  4*4=16
1*5=5  2*5=10  3*5=15  4*5=20  5*5=25
1*6=6  2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7  2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8  2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9  2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81
```

例 3-16 通项公式 $a_n=\frac{1!}{n+1}+\frac{2!}{n+1}+\cdots+\frac{n!}{n+1}(n\geqslant 1)$,求前 n 项和 S_n,n 由键盘

输入。

这题也需要两层循环来实现，内层循环计算通项 a_n 的值，外层循环将内层循环计算好通项 a_n 的值累加起来。

程序如下：

```
#coding:utf-8
#例 3-16  循环嵌套的用法
n=input("input num n")
sum=0                                    #前 n 项和 sum 初始化为 0
for i in range(1,n+1):
    an=0                                 #通项 an 初始化为 0
    x=1                                  #阶乘初始化为 1
    for j in range(1,i+1):
        x=x * j                          #j 的阶乘
        an=an+float(x)/(i+1)             #求通项 an
    sum=sum+an                           #累加通项 an
print "The sum of %d is %f" % (n,sum)
```

如果输入的 n 依次为 2、5、7，输出的结果如下：

```
The sum of 2 is 1.500000
The sum of 5 is 35.850000
The sum of 7 is 899.689286
```

3.5 本章小结

本章主要讲解了以下几个知识点：

(1) 算法及其要素和特性。算法(Algorithm)是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令。算法代表着用系统的方法描述解决问题的策略机制。算法的三要素包括操作、控制结构、数据结构。算法的 5 个基本特征分别是有穷性、确定性、有零个或多个输入、有一个或多个输出、有效性。

(2) 算法的表示方法。包括自然语言、传统流程图、结构化流程图、伪代码等。

(3) 常用的结构化程序设计方法——自上而下求精法。自上而下求精法就是先从总体上分析解决问题的大体步骤，然后对前一次求精结果再进行细化，得到这次的求精结果，如果该结果的求解过程已经很明确，那么就结束，否则，还要进一步求精，直至问题求解过程很明确。

(4) 选择结构。分别讲解了 if、if/else、if/elif/else 这三种选择结构，通过选择结构能够根据条件的不同选择执行哪部分的语句，从而控制程序的流程。

(5) 循环结构。分别讲解了 while、for 这两种循环结构，通过循环结构同样能够控制程序的流程，使某部分语句循环执行。同时还介绍了 continue 语句和 break 语句。

continue 语句用于跳过该次循环，而 break 语句则是用于退出循环。

3.6 习　　题

一、解答题

1. 什么是算法？算法的基本特性有哪些？
2. 算法的表示方法有哪些？
3. 选择结构有哪些？它们的异同点是什么？
4. 循环结构有哪些？它们的异同点是什么？

二、看程序写结果

1.

```
for x in range(1,100):
    if x%9:
        continue
    if x>50:
        break
    print x
```

2.

```
k=4
n=0
while n<k:
    n=n+1
    if n%2==0:
        continue
    k=k-1
print "k=%d, n=%d" % (k,n)
```

3.

```
x=1
y=0
if not x:
    y+=1
elif x==0:
    if x:
        y+=2
    else:
        y+=3
print y
```

4.

```
n=5
sum=0
```

```
for i in range(1,n+1):
    an=0
    for j in range(1,i+1):
        an=an+float(j)/(i+1)
    sum=sum+an
print sum
```

三、上机练习

1. 编写一个程序,输入一个日期,格式为yyyy-mm-dd,输出一个整数,表示该日期在当年中为第几天。例如,输入2015-10-01,输出为274(提示:若输入的数据存到str_date中,则获取年份字符串使用str_date[0:4],获取月份字符串使用str_date[5,7],获取日的字符串使用str_date[8,10])。

2. 编写一个程序,求出1~100之间的素数。

3. 编写一个程序,求出10的阶乘。

4. 斐波那契数列是"1,1,2,3,5,8,13,21,…"。即数列中的下一个值是数列中前两个值的和。编写一个输出斐波那契数列前30个数的程序。

5. 编写程序输出 $S_n=a+aa+aaa+\cdots+aa\cdots a$(n个a)的值,其中a是一个数字,n表示a的位数,例如a=3,n=4时 $S_n=3+33+333+3333$。a、n由键盘输入。

6. 编写程序输出所有的"水仙花数",所谓"水仙花数"是指一个三位数,其各位数字立方和等于该数本身。例如,153是一个水仙花数,因为 $153=1^3+5^3+3^3$。

7. $f(x)=\begin{cases} g(x)+x^2+x+3, & x<g(x) \\ g(x)-x^2, & x\geqslant g(x) \end{cases}$,其中,$g(x)=x^2-750$,$x\in Z$,编写程序输出f(x)的值,x由键盘输入。

第 4 章

序列：字符串、列表和元组

本章学习目标

- 掌握序列类型通用的操作符和内建函数
- 掌握字符串的创建、访问、操作和常用内置函数
- 掌握列表的创建、访问、操作和常用内置函数
- 掌握元组的创建、访问、操作和常用内置函数
- 会利用列表实现堆栈、队列等常用的数据结构

在第 2 章中已经介绍了 Python 的整型、浮点型、布尔类型。本章将介绍序列类型，它们的成员是有序排列的，并且可以通过下标偏移量访问到它们的一个或者多个成员。这种序列类型包括字符串、列表和元组类型。

本章先介绍适用于所有序列类型的操作符和内建函数，然后再针对每种序列类型展开介绍。

4.1 概　　述

4.1.1 序列

序列类型都有着相同的访问模式：通过指定一个下标位移量的方式可以访问到序列中的任何一个元素；通过切片的方式一次可以得到多个元素。下标位移量是从 0 开始到 N－1（N 是序列中元素的个数）结束或者从序列最后一个元素－1 开始到第一个元素－N 结束。图 4-1 描述了序列中元素的存储形式。其中，s 是序列的名称，N 是序列的元素个数。

s[0]		s[−N]
s[1]		s[−(N−1)]
s[2]		s[−(N−2)]
	⋮	
s[N−2]		s[−2]
s[N−1]		s[−1]

图 4-1　序列元素的存储形式

4.1.2 序列类型操作符

表 4-1 列出了所有序列类型都适用的操作符，它们的优先级顺序从高到低。

1. 成员关系操作符（in、not in）

成员关系操作符是用来判断一个元素是否属于一个序列的。例如，对于字符串类型，就是判断一个字符是否属于这个字符串；对于列表类型，则表示一个列表元素是否属于该

列表。in/not in 操作符的返回值一般来说就是 true/false，若某个元素属于一个序列，则返回 true，否则返回 false。

表 4-1　序列类型操作符

序列操作符	作　用
seq[index]	获得下标为 index 的元素
seq[index1：index2]	获得下标从 index1 到 index2 间的元素集合
seq * expr	序列重复 expr 次
seq1＋seq2	连接序列 seq1 和 seq2
element **in** seq	判断 element 元素是否包含在 seq 中
element **not in** seq	判断 element 元素是否不包含在 seq 中

2. 连接操作符（＋）

这个操作符的作用是用来连接两个或多个相同类型的序列。它的语法如下：

```
sequence1+ sequence2
```

该表达式的返回值是一个包含了 sequence1 和 sequence2 的序列。注意，这个操作不是最快或者说是最有效的。例如，对于字符串，通过把所有的字符串放到一个列表或者可迭代的对象中，然后调用 join 函数把所有的内容连接在一起的方式更节约内存；同样的，对于列表，使用列表类型的 extend 函数把两个或多个列表对象合并起来的效率更高。当需要简单地把两个对象的内容合并，或者说不能通过序列的内建函数来完成时，连接操作符还是一个不错的选择。

3. 重复操作符（*）

重复操作符可以使某个序列重复多次，语法如下：

```
sequence * expr
```

expr 是一个表达式，这个表达式为整型。返回值为一个包含了多份原序列的拷贝的新序列。

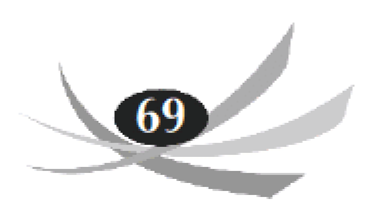
4. 切片操作符（[]、[:]、[：：]）

因为序列类型是由一些元素共同组成的一个有序的整体，所以，可以用方括号加一个下标的方式访问它的每一个元素，或者通过在方括号中用冒号把开始下标和结束下标分开的方式来访问一组连续的元素。这种访问序列的方式就叫做切片。

访问某一个元素的语法如下：

```
sequence[index]
```

sequence 是序列的名称，index 是想要访问的元素对应的偏移量。偏移量可以是正值（正索引）。范围从 0 到 N－1，N 为序列长度（或序列的元素个数），可以通过内建函数 len()得到 N。此外，偏移量还可以是负值（负索引），范围从－1 到序列长度的负值。所



以，index 的返回可以是 0≤index≤len(sequence)－1 或者是－len(sequence)≤index≤－1。正负索引的区别在于正索引以序列的开头为起点，负索引以序列的结束为起点。注意，如果索引超出上述的范围，程序会抛出 IndexError 异常，关于异常的知识将在第 9 章中介绍。

上述介绍了访问某一个元素的方法，如果想一次得到多个元素，可以指定开始索引和结束索引，并且以冒号分隔，其语法如下：

```
sequence[starting_index:ending_index]
```

通过这种方式可以得到从开始索引到结束索引(不包括结束索引对应的元素)之间的一组连续的元素。起始索引和结束索引都是可选的，如果没有指定，切片操作会从序列的最开始处开始，或者直到序列的最末端结束。

其实，除了可以指定开始索引和结束索引，还可以指定步长索引，其语法如下：

```
sequence[starting_index:ending_index:step_index]
```

这里的步长索引类似于 C 语言中的 for 语句的步长参数，起始索引和结束索引也都是可选的。

注意，开始索引可以小于 0，而结束索引可以大于索引长度。

4.1.3　序列类型内建函数

1. 类型转换函数

内建函数 list()、str()、tuple()被用做在各种序列类型之间转换。可以把它们理解成其他语言中的类型转换。其实，这里的类型转换并没有进行任何的转换，只是把其内容复制到新生成的对象中。表 4-2 列出了主要的序列类型的转换函数。

表 4-2　主要的序列类型的转换函数

函　　数	功　　能
list(iter)	把可迭代对象转化为列表
str(obj)	把 obj 对象转换成字符串
unicode(obj)	把 obj 对象转换成 Unicode 字符串(使用默认编码)
tuple(iter)	把一个可迭代对象转换成一个元组对象
type(obj)	函数返回的是 obj 对象的类型

2. 可操作函数

表 4-3 列出了序列类型可操作的内建函数。需要注意的是：len()、reverse()和 sum()函数只能接受序列类型对象作为参数，而剩下的则可以接受可迭代对象作为参数，另外，max()和 min()函数也可以接受一个参数列表。

表 4-3 序列类型可操作的内建函数

函　　数	功　　能
enumerate(iter)	接受一个可迭代对象作为参数,返回一个 enumerate 对象(同时也是一个迭代器),该对象生成由 iter 中每个元素的 index 值和 item 值组成的元组
len(seq)	返回 seq 的长度
max(iter, key=None) or max(arg0,arg1,…,key=None)	返回 iter 或(arg0,arg1,…)中的最大值,如果指定了 key,这个 key 必须是一个可以转给 sort()方法的,用于比较的回调函数
min(iter, key=None) or min(arg0,arg1,…,key=None)	返回 iter 或(arg0,arg1,…)中的最小值,如果指定了 key,这个 key 必须是一个可以转给 sort()方法的,用于比较的回调函数
reverse(seq)	接受一个序列作为参数,返回一个逆序的迭代器
sorted(iter, func=None, key=None, reverse=False)	接受一个可迭代对象作为参数,返回一个有序的列表;如果指定 func 和 key 参数,则按照指定的方式比较各个元素,如果 reverse 置为 true,则列表以反序排列,默认为 false(正序)
zip([iter0,iter1,…,iterN])	返回一个列表,其第一个元素是 iter0、iter1、…这些元素的第一个元素组成的一个元组,第二个…以此类推

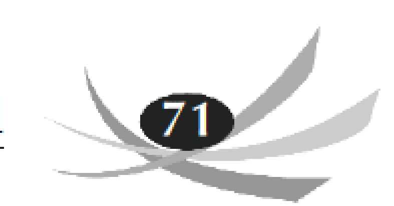
接下来将分别介绍这些序列类型。

4.2 字　符　串

4.2.1 创建字符串

字符串类型是 Python 语言中很常见的类型,并且是一种不可变的序列类型。我们可以通过在引号(包括单引号'、双引号"或三个引号''')间包含字符的方式或者通过内建函数 str()的方式创建字符串。例如'python'、"hello"、'''world'''、str(range(5))等等。注意,"或"等这些引号本身只是一种表示方式,不是字符串的一部分。因此,字符串'python'只有 p、y、t、h、o、n 这 6 个字符。

```
#例 4-1  创建字符串
>>>string1='Hello World!'                  #使用单引号
>>>string2="I want to learn Python."       #使用双引号
>>>string3='''Python is cool!'''           #使用三引号
>>>string4=str(range(5))                   #使用 str()函数
>>>print string1
Hello World!                               #使用 print 语句,输出的字符无引号
>>>string1
'Hello World!'                             #不使用 print 语句,输出的字符带引号
>>>print string2
I want to learn Python.
>>>print string3
```




```
Python is cool!
>>>print string4
[0,1,2,3,4]
```

从上面可以看出，通过引号的这三种方式输出的效果都是一样的，那为什么要定义这么多的方式呢？这主要是为了使用更灵活方便。请看例4-2。

如果字符串中包含单引号'或双引号"，可以用两种处理方法：

（1）如果字符串中包含单引号'(双引号")可以使用双引号"(单引号')或者三个引号'''括起来，例如"I'm OK"，'He said："Welcome to learn python"'。甚至有'''He said："I'm OK"'''。

```
#例 4-2　字符串的使用
>>>print "I'm OK"
I'm OK
>>>print 'He said: "Welcome to learn python"'
He said: "Welcome to learn python"
>>>print '''He said:"I'm OK"'''
He said:"I'm OK"
```

（2）使用转义字符"\"来表示。例如'He said:\"I\'m OK\"'。当使用单引号'括起来时，对字符串中的单引号'必须用转义字符，而双引号"，可以使用转义字符，也可以不用，反之亦然。

```
#例 4-3　转义字符的使用 1
>>>print 'He said:"I\'m OK"'
He said:"I'm OK"
```

转义字符就是以一个字符"\"开头的字符序列。常用的以"\"开头的特殊字符见表4-4。

表4-4　转义符及其作用

字符形式	含　义	ASCII
\n	换行，将当前位置移动到下一行开头	10
\t	水平制表(跳到下一个 Tab 位置)	9
\b	退格，将当前位置移动到本行开头	8
\r	回车，将当前位置移动到本行开头	13
\f	换页，将当前位置移动到下一页开头	12
\\	代表一个反斜杠字符"\"	92
\'	代表一个单引号字符	39
\"	代表一个双引号字符	34
\ddd	1到3位八进制数所代表的字符	
\xhh	1到2位十六进制数所代表的字符	

表 4-4 中列出的字符称为“转义字符”，意思是将反斜杠“\”后面的字符转成另外的意义。如“\b”中的 b 不代表字母 b 而作为退格符。

表 4-4 中最后一行是一个十六进制数的 ASCII 码表示的一个字符，例如“\x4F”代表十六进制 ASCII 码为 4F 的字符‘O’，十六进制 4F 相当于十进制 79，从 ASCII 码表可以看到 ASCII 码(十进制)为 79 的字符是大写字母 O。用表 4-4 中的方法可以表示任何可输出的字母字符、专用字符、图形字符和控制字符。

```
#例 4-4  转义字符的使用 2
>>>print 'abc\bd\tefg\n'
abd     efg

>>>
```

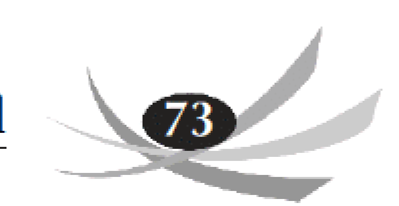
程序先输出 abc，此时光标移到第 4 列准备输出下一个字符，然后遇到“\b”，它的作用是“退一格”，光标移回到第 3 列，接着输出 d，注意，此时输出的 d 已经把刚才在第 3 列输出的 c 覆盖了，再然后遇到“\t”，它的作用是“跳格”，即跳到下一个“制表位置”，在所有系统中一个“制表区”占 8 列。“下一个制表位置”从第 9 列开始，输出 efg，最后再换行。

4.2.2 访问字符串

这里的值包括一个字符或多个字符组成的字符串(子串)。注意，Python 语言没有字符类型，而是用长度为 1 的字符串来表示这个概念。其实，这也是一个子串。

在 4.1.2 节中，我们介绍了通过切片操作的方式来访问序列的一个或一组元素，接下来我们针对字符串这种序列类型举例说明如何访问字符串的值。

```
#例 4-5  获取字符串的字串
>>>a='Welcome'
#正索引
>>>print "a[0]:", a[0]
a[0]:W
>>>print len(a)                           #通过 len()函数获取字符串长度
7
>>>print "a[7]:", a[7]
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
IndexError: string index out of range
>>>print "a[1:5]:", a[1:5]
a[1:5]:elco
>>>print "a[2:7]:", a[2:7]
a[2:7]:lcome
>>>print "a[2:7:2]:", a[2:7:2]            #使用步长索引
a[2:7:2]:loe
>>>print "a[2:10]:", a[2:10]
a[2:10]:lcome
```



```
#负索引
>>>print "a[-1]:", a[-1]
a[-1]:e
>>>print "a[-3:-1]:", a[-3:-1]
a[-3:-1]:om

#默认索引
>>>print "a[1:]:", a[1:]
a[1:]:elcome
>>>print "a[-3:]:", a[-3:]
a[-3:]:ome
>>>print "a[:4]:", a[:4]
a[:4]:Welc
>>>print "a[:]:", a[:]
a[:]:Welcome
```

从这个例子可以看到，访问第i(i从0开始)个字符，可以通过a[i]来访问。正索引，其范围为0～len(a)－1，负索引范围为－len(a)～－1。访问单个字符时，如果超出这个范围就会抛IndexError异常(string index out of range)，提示字符串索引超出范围。对于要获取字符串变量a中连续的字符，则可以通过a[i：j]获取第i到j个连续的字符(不包含第j个字符)。如输出a[1：5]就得到“elco”，不包含第5个字符“m”。这里的索引即使超出字符串的长度，也不会报错。如输出a[2：10]，则会输出“lcome”，此外，还可以指定步长索引来间隔获取字符。如输出a[2：7：2]，则会间隔一个字符输出索引为第2到第7个中的偶数索引的字符，即输出“loe”。如果i、j是负索引，则从字符串右边向左数起，即－1为字符串从右边向左数起的第一个字符，如输出a[－3：－1]就得到“om”，不包含“e”。当不指定开始索引时，其默认索引为0，如输出a[：4]，则会依次输出索引0到4(不包括4)对应的字符，即输出“Welc”。如不指定结束索引，其默认索引为len(a)(包括字符串的最后一个字符)。如果开始索引和结束索引都不指定，则会输出整个字符串。

4.2.3 字符串操作符

在这一小节，我们将结合例子来说明字符串的操作符。

```
#例4-6 字符串操作符
>>>a="Hello"
>>>b="World!"

#1.连接操作符(+)
>>>a+b
'HelloWorld!'
>>>a+' '+b
'Hello World!'
>>>print '字符串a的字符个数为:'+len(a)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
TypeError: cannot concatenate 'str' and 'int' objects

#2.重复操作符(*)
>>>a*3
'HelloHelloHello'

#3.成员关系操作符(in、not in)
>>>'H' in a
true
>>>'h' in a
false
>>>'H' not in a
false
>>>'h' not in a
true

#4.原始字符串操作符(r/R)
>>>print r'\n\bdf\t\n'                    #有原始字符串操作符 r,原样输出
\n\bdf\t\n
>>>print 'd\tf\ng'                        #无原始字符串操作符 r/R,转义字符起作用
d       f
g
```

说明：

(1) 使用连接操作符时，只能把两个或多个字符串连接起来，当连接操作符左右两边的类型不一致时会抛 TypeError 异常。如将字符串"字符串 a 的字符个数为："和 len(a) 连接时就抛 TypeError 异常，提示字符串类型对象不能和整型对象做连接操作。

(2) 原始字符串操作符的作用是让给定的字符串原样赋值或输出，即字符串中含有的转义字符将不起作用，如 print r'\n\bdf\t\n'就直接输出字符串"\n\bdf\t\n"。

其实，在 4.2.2 小节里介绍的切片操作也是一个操作符。

表 4-5 列出了 Python 提供的常用字符串操作符。

表 4-5　Python 常用的字符串操作符

操　作　符	描　　述
+	字符串连接
*	重复原样字符串
[]	通过索引获取字符串中的单个字符
[:]	截取字符串中的子串
in	成员关系操作符，如果字符串中包含给定的字符，则返回 true，否则返回 false

续表

操　作　符	描　　述
not in	成员关系操作符，如果字符串中不包含给定的字符，则返回 true，否则返回 false
r/R	原始字符串，所有的字符串都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。原始字符串除在字符串的第一个引号前加上字母"r"(可以大小写)以外，与普通字符串有着几乎完全相同的语法

4.2.4 常用字符串内建函数

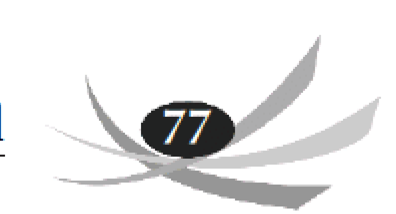
Python 语言之所以强大是因为其提供了非常丰富的各种类库(关于类的概论将在第7章中讲解)和函数。表 4-6 列出了常用的字符串函数。

表 4-6　常用的字符串函数

方　　法	描　　述
string. capitalize()	把字符串的第一个字符大写
string. center(width)	返回一个原字符串居中，并使用空格填充至长度 width 的新字符串
string. count(str, beg=0, end=len(string))	返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定，则返回指定范围内 str 出现的次数
string. decode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式解码 string，如果出错默认报一个 ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
string. encode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式编码 string，如果出错默认报一个 ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
string. endswith(str, beg=0, end=len(string))	检查字符串是否以 str 结束，如果 beg 或者 end 指定范围，则检查指定的范围内是否以 str 结束，如果是，返回 true，否则返回 false
string. expandtabs(tabsize=8)	把字符串 string 中的 tab 符号转为空格，默认的空格数 tabsize 是 8
string. find(str, beg=0, end=len(string))	检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回－1
string. index(str, beg=0, end=len(string))	跟 find()方法一样，只不过如果 str 不在 string 中会报一个异常
string. isalnum()	如果 string 至少有一个字符并且所有字符都是字母或数字，则返回 true，否则返回 false
string. isalpha()	如果 string 至少有一个字符并且所有字符都是字母则返回 true，否则返回 false
string. isdecimal()	如果 string 只包含十进制数字则返回 true，否则返回 false
string. isdigit()	如果 string 只包含数字，则返回 true，否则返回 false

续表

方　　法	描　　述
string. islower()	如果 string 中包含至少一个区分大小写的字符,并且所有这些(区分大小写的)字符都是小写,则返回 true,否则返回 false
string. isnumeric()	如果 string 中只包含数字字符,则返回 true,否则返回 false
string. isspace()	如果 string 中只包含空格,则返回 true,否则返回 false
string. istitle()	如果 string 是标题化的(见 title())则返回 true,否则返回 false
string. isupper()	如果 string 中包含至少一个区分大小写的字符,并且所有这些(区分大小写的)字符都是大写,则返回 true,否则返回 false
string. join(seq)	Merges (concatenates)以 string 作为分隔符,将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
string. ljust(width)	返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串
string. lower()	转换 string 中所有大写字符为小写
string. lstrip()	截掉 string 左边的空格
string. maketrans(intab, outtab)	maketrans()方法用于创建字符映射的转换表,对于接受两个参数的最简单的调用方式,第一个参数是字符串,表示需要转换的字符,第二个参数也是字符串,表示转换的目标
max(str)	返回字符串 str 中最大的字母
min(str)	返回字符串 str 中最小的字母
string. partition(str)	有点像 find()和 split()的结合体,从 str 出现的第一个位置起,把字符串 string 分成一个 3 元素的元组(string_pre_str,str,string_post_str),如果 string 中不包含 str,则 string_pre_str==string
string. replace(str1, str2, num=string. count(str1))	把 string 中的 str1 替换成 str2,如果 num 指定,则替换不超过 num 次
string. rfind(str, beg=0, end=len(string))	类似于 find()函数,不过是从右边开始查找
string. rindex(str, beg=0,end=len(string))	类似于 index(),不过是从右边开始
string. rjust(width)	返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串
string. rpartition(str)	类似于 partition()函数,不过是从右边开始查找
string. rstrip()	删除 string 字符串末尾的空格
string. split (str = "", num = string. count(str))	以 str 为分隔符切片 string,如果 num 有指定值,则仅分隔 num 个子字符串
string. splitlines (num = string. count('\n'))	按照行分隔,返回一个包含各行作为元素的列表,如果 num 指定则仅切片 num 个行



续表

方　　法	描　　述
string. startswith(str, beg=0, end=len(string))	检查字符串是否以 str 开头,是则返回 true,否则返回 false。如果 beg 和 end 指定值,则在指定范围内检查
string. strip([obj])	在 string 上执行 lstrip()和 rstrip()
string. swapcase()	翻转 string 中的大小写
string. title()	返回"标题化"的 string,就是说所有单词都是以大写开始,其余字母均为小写(见 istitle())
string. translate(str, del="")	根据 str 给出的表(包含 256 个字符)转换 string 的字符,要过滤掉的字符放到 del 参数中
string. upper()	转换 string 中的小写字母为大写
string. zfill(width)	返回长度为 width 的字符串,原字符串 string 右对齐,前面填充 0
string. isdecimal()	isdecimal()方法检查字符串是否只包含十进制字符。这种方法只存在于 unicode 对象

这里举例讲解其中的几个函数。

```
#例 4-7　常用字符串内建函数
>>>a="welcome"
>>>print a.capitalize()          #capitalize()函数
Welcome
>>>b="hello world,hello python,hello C"
>>>print b.count("hello")        #count(str)函数
3
>>>print b.count("hello",1)      #count(str,beg[,end])函数
2
>>>print b.endswith("hello")     #endswith(str)函数
false
>>>print b.endswith("C")         #endswith(str)函数
true
>>>print b.find("python")        #find(str)函数
18
>>>print b.find("Python")        #find(str)函数,注意这里的 Python 第一个字母大写
-1
>>>c="HELLO WORLD"
>>>print c.lower()               #lower()函数
hello world
```

4.3　列　　表

像字符串类型一样,列表类型也是序列数据类型,也可以通过下标或者切片操作的方式来访问某一个或者某一块连续的元素。但列表又有很多不同于字符串的特性。列表是一种

可变序列类型,而字符串是一种不可变序列类型;列表可以包含任意数量,任意类型的Python对象,而字符串只能由字符组成,而且不能试图改变它的某个值。相比于C语言的数组(所有元素只能是一种类型,且数组大小不能动态改变),Python的列表更加灵活。

4.3.1 创建列表

创建一个列表,可以通过使用方括号,并把方括号里的每一个元素采用逗号进行分隔或者使用内建函数list()来实现。列表的每一个元素可以是不同的数据类型。列表中的元素还可以是列表。

创建列表的一般格式如下:

```
list_name=[element1,element2,element3,...,elementN]
```

```
#例 4-8  创建列表
#通过方括号和逗号来创建列表
>>>num_list=[1,2,3,4,5]                    #元素都是整型
>>>string_list=['a','b','c']               #元素都是字符串
#元素类型有整型、浮点型、字符串类型和布尔型
>>>mix_list=[123,4.56,'abc',True]
#元素还可以是列表类型
>>>list_in_list=[num_list,string_list]
#通过内建函数 list()来创建列表
>>>another_num_list=list(range(5))
>>>print num_list
[1,2,3,4,5]
>>>print string_list
['a','b','c']
>>>print mix_list
[123,4.56,'abc',True]
>>>print list_in_list
[[1,2,3,4,5],['a','b','c']]
>>>print another_num_list
[0,1,2,3,4]
```

4.3.2 访问列表

和访问字符串类似,访问列表也是通过切片操作来实现的。不过列表的切片操作返回的是一个对象或者多个对象的集合,而不是像字符串那样,返回一个字符或者一个子串。

列表的切片操作也遵循正负索引规则,也有开始索引和结束索引,如果这两个索引都没指定,默认也会分别指到序列的开始和结束位置。

```
#例 4-9  访问列表
>>>num_list=[1,2,3,4,5]                    #元素都是整型
```

```
>>>string_list=['a','b','c','d']          #元素都是字符串
#元素类型有整型、浮点型、字符串类型、布尔型和列表类型
>>>mix_list=[123,4.56,'abc',True,[789,'789','efg']]
>>>num_list[1]
2
>>>num_list[1:3]
[2,3]
>>>num_list[1:]
[2,3,4,5]
>>>string_list[:3]
['a','b','c']
>>>string_list[-3:-1]
['b','c']
>>>mix_list[4]
[789,'789','efg']
>>>mix_list[-3:]
['abc',True,[789,'789','efg']]
```

访问二维列表可以通过如下方式：

```
list_name[index1][index2]
```

其中，index1为二维列表的元素索引，index2为二维列表中index1索引指向的元素中的元素索引。访问方式类似于其他语言(如C语言)访问二维数组。下面通过一个例子来理解访问二维列表。

```
#例 4-10  访问二维列表
#创建列表 list1 和 list2
list1=['1001','1002','1003']
list2=['1004','1005','1006']
#列表 student_list 由列表 list1 和 list2 组成
student_list=[list1,list2]
#输出列表 student_list 中的第一个元素(list1)中的第一个元素的值('1001')
print 'student_list[0][0]=',student_list[0][0]
#输出列表 student_list 中的第一个元素(list1)中的第二个元素的值('1002')
print 'student_list[0][1]=',student_list[0][1]
#输出列表 student_list 中的第二个元素(list2)中的第三个元素的值('1006')
print 'student_list[1][2]=',student_list[1][2]

#遍历 student_list 二维列表
for i in range(len(student_list)):
    for j in range(len(student_list[i])):
        print 'student_list['+str(i)+']['+str(j)+']=',student_list[i][j]
```

该程序运行的结果如下：

```
student_list[0][0]=1001
student_list[0][1]=1002
student_list[1][2]=1006
遍历 student_list 二维列表:
student_list[0][0]=1001
student_list[0][1]=1002
student_list[0][2]=1003
student_list[1][0]=1004
student_list[1][1]=1005
student_list[1][2]=1006
```

4.3.3 更新列表

列表是可变数据类型,即列表的长度和元素的值都是可以改变的。下面将介绍更新列表的方式:添加元素、修改元素和删除元素。

1. 添加元素

向列表添加一个元素可以通过 append()函数实现,这个函数用于在列表的尾部添加一个元素。该函数的声明如下:

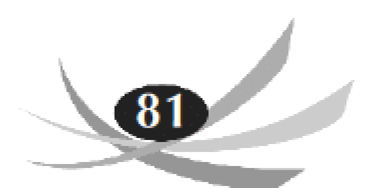
```
list_name.append(value)
```

其中,list_name 为列表名,参数 value 的类型是对象类型,即可以为列表添加任何类型的元素。下面通过一个例子来说明如何使用 append()函数向一个列表添加元素。

```
#例 4-11  使用 append()函数向列表添加一个元素
#定义一个含有 5 个元素(学号)的列表
student_list=['1001','1002','1003','1004','1005']
#使用 len()函数获取 student_list 列表中初始的个数,str()将整型转为字符串类型
print '目前有'+str(len(student_list))+'个学生'
print '刚来了一个学生'
#使用 append()函数向 student_list 列表尾部添加一个 1006 的元素
student_list.append('1006')
#再次输出此时 student_list 列表的长度
print '现在有'+str(len(student_list))+'个学生,他们的学号分别是:'
#使用 for 循环遍历这个 student_list 列表,分别输出这些元素
for item in student_list:
    print item
```

该程序运行的结果如下:

```
目前有 5 个学生
刚来了一个学生
现在有 6 个学生,他们的学号分别是:
1001
1002
```




```
1003
1004
1005
1006
```

此外，Python 还提供了另外一个 insert()函数，这个函数用于将元素插入到列表中的指定索引位置，该函数的声明如下：

```
list_name.insert(index,value)
```

其中，list_name 为列表名，第一个参数 index 为将要插入的列表中元素位置的索引，第二个参数 value 为要插入的值。下面通过一个例子来了解该函数与 append()函数之间的区别。

```
#例 4-12　使用 insert()函数向列表的指定位置插入元素
#定义一个含有 5 个元素(学号)的列表
student_list=['1001','1002','1003','1004','1005']
print '目前有'+str(len(student_list))+'个学生'
print '刚来了一个学生'
#使用 insert()函数向 student_list 列表中索引为 3 的位置插入一个 1006 的元素
student_list.insert(3,'1006')
#再次输出此时 student_list 列表的长度
print '现在有'+str(len(student_list))+'个学生,他们的学号分别是:'
#使用 for 循环遍历这个 student_list 列表,分别输出这些元素
for item in student_list:
    print item
```

该程序运行的结果如下：

```
目前有 5个学生
刚来了一个学生
现在有 6个学生,他们的学号分别是:
1001
1002
1003
1006
1004
1005
```

2. 修改元素

修改列表中的元素也相当简单，只需使用前面介绍的赋值符号和指明需要修改的元素的位置即可。修改列表元素的语法如下：

```
list_name[index]=value
```

其中，index 是列表中的索引，其范围是 0～len(list_name)－1，如果给定的 index 不在此范围，则修改列表操作将失败。value 为元素的值，该值可以是任意的类型。

```
#例 4-13 修改列表的元素
#定义一个含有 5 个元素 (学号)的列表
student_list=['1001','1002','1006','1004','1005']
#输出修改前的列表
print '初始化的 student_list 列表为:'+str(student_list)
#修改列表中索引为 2 的元素,将其改为'1003'
student_list[2]='1003'
#输出修改后的列表
print '修改后的 student_list 列表为:'+str(student_list)
```

该程序运行的结果如下:

```
初始化的 student_list 列表为:['1001','1002','1006','1004','1005']
修改后的 student_list 列表为:['1001','1002','1003','1004','1005']
```

此外,还可以一次性修改(或添加)多个元素,如下面的例子所示:

```
#例 4-14 一次性修改 (或添加)多个元素
#定义一个含有 5 个元素 (学号)的列表
student_list=['1001','1003','1006','1002','1005']
#输出修改前的列表
print '初始化的 student_list 列表为:'+str(student_list)
#修改列表中索引为 1 到 4 的元素,将其分别改为'1002'、'1003'、'1004'
student_list[1:4]=['1002','1003','1004']
#输出修改后的列表
print '修改后的 student_list 列表为:'+str(student_list)

#重新对 student_list 赋值,使其只含有 2 个元素 (学号)
student_list=['1001','1005']
#输出插入前的列表
print '重新初始化的 student_list 列表为:'+str(student_list)
#通过切片方式在索引为 1 的位置上插入多个元素
student_list[1:1]=['1002','1003','1004']
#输出插入后的列表
print '插入后的 student_list 列表为:'+str(student_list)
```

该程序运行的结果如下:

```
初始化的 student_list 列表为:['1001','1003','1006','1002','1005']
修改后的 student_list 列表为:['1001','1002','1003','1004','1005']
重新初始化的 student_list 列表为:['1001','1005']
插入后的 student_list 列表为:['1001','1002','1003','1004','1005']
```

3. 删除元素

删除列表中的元素可以通过 remove()函数实现,该函数用于删除列表中指定值的第一个匹配的元素,其声明如下:

```
list_name.remove(value)
```

其中，参数 value 表示要删除的列表中指定的元素值。注意，该函数只删除匹配的第一个元素值，如果有多个匹配的元素，可以多次调用该函数，以删除所有匹配的元素。此外，如果要删除元素值在列表中不存在，程序将会抛 ValueError 异常。

删除元素还可以使用 del 语句来实现，该语句将删除列表中指定索引位置所对应的元素，该语句的声明如下：

```
del list_name[index]
```

其中，index 表示将要删除的元素所对应的索引。

下面通过一个例子来理解 remove()函数和 del 语句的使用。

```
#例 4-15　删除元素
#定义一个含有 5 个元素(学号)的列表
student_list=['1001','1002','1002','1004','1005']
#输出删除前的列表
print '初始化的 student_list 列表为:'+str(student_list)
#使用 remove()函数将列表中值为'1002'的元素(2 个)都删除
student_list.remove('1002')
#需要执行两次 remove()函数
student_list.remove('1002')
#输出删除后的列表
print '使用 remove()函数删除两个元素后的 student_list 列表为:'+str(student_list)
#使用 del 语句将列表中值为'1004'的元素删除
del student_list[1]
#输出删除后的列表
print '使用 del 语句删除索引为 1 的元素后的 student_list 列表为:'+str(student_list)
```

该程序运行的结果如下：

```
初始化的 student_list 列表为:['1001','1002','1002','1004','1005']
使用 remove()函数删除两个元素后的 student_list 列表为:['1001','1004','1005']
使用 del 语句删除索引为 1 的元素后的 student_list 列表为:['1001','1005']
```

4.3.4　列表操作符

在这一小节，我们将结合例子来说明列表的操作符。

```
#例 4-16　列表操作符
>>>num_list=[12,-34,5.6,7.89e1]
>>>string_list=['I','want','to','learn','Python']
>>>mix_list=[1.2,'welcome',True,[3.4,'C']]

#1.连接操作符(+)
>>>num_list+string_list
```

```
[12,-34,5.6,78.9,'I','want','to','learn','Python']
>>>num_list+mix_list
[12,-34,5.6,78.9,1.2,'welcome',True,[3.4,'C']]
>>>string_list+mix_list
['I','want','to','learn','Python',1.2,'welcome',True,[3.4,'C']]
>>>string_list+'me too'
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
TypeError: can only concatenate list (not "str") to list

#2.重复操作符(*)
>>>num_list*2
[12,-34,5.6,78.9,12,-34,5.6,78.9]
>>>num_list*3
[12,-34,5.6,78.9,12,-34,5.6,78.9,12,-34,5.6,78.9]

#3.成员关系操作符(in、not in)
>>>7.89e1 in num_list
true
>>>'Python' in string_list
true
>>>'welcome' in mix_list
true
>>>'C' in mix_list
false
>>>'C' not in mix_list
true
>>>'C' in mix_list[3]
true
```

说明：

(1) 使用连接操作符时，只能把两个或多个列表连接起来，当连接操作符左右两边的类型不一致时会抛 TypeError 异常。如将 string_list 列表和字符串'me，too'连接时就抛 TypeError 异常，提示列表只能连接列表类型。

(2) 成员关系操作符判断的是一级成员关系，而不是二级或多级成员关系。因为'C'不是 mix_list 列表的一级成员关系，所以'C' in mix_list 返回 false，但'C'是 mix_list[3]([3.4,'C'])列表的一级成员关系，所以'C' in mix_list[3]返回 true。

4.3.5 常用列表内建函数

1. 适用于列表的常用内建函数

表 4-7 列出了可以应用于列表的常用内建函数。

表 4-7　可以应用于列表的常用内建函数

函　　数	功能（注意，参数的内容并没有改变）
cmp(list1,list2)	比较两个列表中的元素，如果 list1 中的元素比 list2 中对应的元素大，则函数返回 1；如果小，则函数返回 −1；如果 list1 和 list2 中所有的元素对应相等，则函数返回 0
len(list)	函数返回 list 列表的元素个数
max(list)	函数返回 list 列表中元素最大的值
min(list)	函数返回 list 列表中元素最小的值
sorted(list)	函数返回一个列表，该列表按从小到大的顺序排好
reversed(list)	函数返回已翻转的迭代器，翻转的意思是将第一个元素和最后一个元素对调，第二个元素和倒数第二个元素对调，以此类推
sum(list)	该函数只适用于列表中元素都是数字的时候，函数返回列表中所有数字的代数和
list(seq)	该函数返回的是一个列表，其元素来自于序列
type(obj)	函数返回的是 obj 对象的类型

下面将结合例子介绍其中的一些函数。

（1）cmp()函数

使用 cmp()函数比较两个列表 list1 和 list2 时，将对这两个列表逐项比较，首先比较第一个元素，如果 list1 中的元素大于 list2 中的元素，则停止比较，函数返回 1；如果小于，也停止比较，函数返回 −1；如果相等，则继续比较下一个元素，直到出现不相等的元素，或者到达较短列表的最后一个元素。此时，列表长的被认为是“较大”的。还有一种情况就是两个列表的所有元素对应相等，此时，两个列表相等，函数返回 0。

比较时如果两个元素的类型相同，则比较它们的值（字符串中字符的比较是比较它们的 ASCII 码的大小，而不是字母的顺序）；如果不相同，则检查它们是否是数字（包括整型、浮点型）。如果是数字，执行必要的强制类型转换，然后再比较；如果有一方是数字，则另一方的元素“大”（数字被认为是“最小的”）；否则，通过类型名字的字母顺序进行比较。

```
#例 4-17　cmp()函数的使用
>>>num1_list=[1.23,-4,5.6,7.89e1]
>>>num2_list=[1.23,2.0,5.6,7.89e1]
>>>num3_list=[1.23,-4,5.6,7.89e1]
>>>string1_list=['I','want','to','learn','Python','and','Java']
>>>string2_list=['I','want','to','learn','Python']
#第一个元素相等,比较第二个元素,将-4转换成浮点数-4.0,小于 2.0,所以返回-1
>>>cmp(num1_list,num2_list)
-1
#两个列表中的所有元素都相等,所以返回 0
```

```
>>>cmp(num1_list,num3_list)
0
#类型不一致,数字类型小于字符串类型,所以返回-1
>>>cmp(num1_list,string1_list)
-1
#前 4 个元素都分别相等,此时列表 string2_list 已到末尾,所以返回 1
>>>cmp(string1_list,string2_list)
1
```

(2) sorted()函数和 reversed()函数

sorted()函数的作用是返回已排好序的列表,其元素是从小到大(这里的大小比较使用 cmp()函数的比较规则)排序的。reversed()函数的作用是返回已翻转的迭代器。所谓翻转就是将列表中的元素进行翻转,即第一个元素和最后一个元素对调,第二个元素和倒数第二个元素对调,以此类推。

```
#coding:utf-8
#例 4-18  sorted()函数和 reversed()函数的使用
#定义一个混合类型列表
mix_list=['I',12,'want','to',3.4,'learn','Python']
#输出初始化列表
print '初始化列表为:',mix_list
#输出从小到大的顺序列表
print '排序后的列表为:',sorted(mix_list)
#使用 for 循环依次输出翻转后的迭代器中的元素
print '翻转后的迭代器中的元素:',list(reversed(mix_list))
```

程序运行结果如下:

```
初始化列表为: ['I', 12, 'want', 'to', 3.4, 'learn', 'Python']
排序后的列表为: [3.4, 12, 'I', 'Python', 'learn', 'to', 'want']
翻转后的迭代器中的元素为: ['Python', 'learn', 3.4, 'to', 'want', 12, 'I']
```

2. 列表对象的常用内建函数

表 4-8 列出了列表对象的常用内建函数。其中的一些函数已在前面介绍过。注意,表中的函数将会改变列表 list 的内容。例如,list. reverse()将列表中的元素进行翻转。前面介绍的 sorted(list)函数返回翻转的列表,但列表 list 并没有发生变化。

表 4-8 列表对象的常用内建函数

函　　数	功　　能
list. append(obj)	向列表中添加一个对象 obj
list. count(obj)	返回对象 obj 在列表中出现的次数
list. extend(seq)	把序列 seq 的内容添加到列表中

续表

函　　数	功　　能
list.index(obj,i=0,j=len(list))	返回 list[k]==obj 的 k 值,且 k 的范围为 i≤k<j;否则抛 ValueError 异常
list.insert(obj)	在索引 index 的位置插入对象 obj
list.pop(index=-1)	删除指定索引的对象,并将该对象返回,默认是最后一个对象
list.remove(obj)	删除匹配的第一个元素值,如果有多个匹配的元素,可以多次调用该函数,以删除所有匹配的元素
list.reverse()	将列表中的元素进行翻转
list.sort(func=None,key=None,reverse=false)	以指定的方式排序列表中的元素,如果 func 和 key 参数指定,则按照指定的方式比较各个元素;如果 reverse 赋值为 true,则列表以反序(从大到小)排序

下面举例介绍其中的几个函数。

```
#coding:utf-8
#例 4-19　列表对象的常用内建函数
#定义一个含有 3个元素 (学号)的列表
student_list=['1001','1002','1002']
#输出初始化列表
print "初始化列表:",student_list
print "'1002'在列表中出现的次数:",student_list.count('1002')
#定义另外一个列表
another_student_list=['1003','1004']
#将 another_student_list 列表的内容添加到 student_list 列表中,注意该函数返回 None
student_list.extend(another_student_list)
#输出添加 another_student_list 列表后的列表
print "添加另一个列表后的列表:",student_list
print "'1003'在列表中的索引:",student_list.index('1003')
#删除最后一个元素
student_list.pop()
print "删除最后一个元素后的列表:",student_list
#删除索引为 1的元素
student_list.pop(1)
print "删除索引为 1的元素后的列表:",student_list
```

程序运行结果如下:

```
初始化列表: ['1001', '1002', '1002']
'1002'在列表中出现的次数: 2
添加另一个列表后的列表: ['1001', '1002', '1002', '1003', '1004']
'1003'在列表中的索引: 3
删除最后一个元素后的列表: ['1001', '1002', '1002', '1003']
```

删除索引为 1 的元素后的列表：['1001', '1002', '1003']

3. 列表的应用——实现堆栈和队列

堆栈和队列是数据结构中比较常见的两种结构。如函数调用就需要用到堆栈这种数据结构;CPU 的资源需要用到队列这种数据结构,根据用户请求的先后顺序来分配资源。

堆栈是限定只能在表的一端进行插入和删除的线性表。在表中允许插入和删除的一端叫做栈顶(top);表的另一端则叫做栈底(bottom)。栈又称后进先出(Last In First Out,LIFO)表。

队列是一种特殊的线性表。在队列中,仅允许一端进行插入,在另一端进行删除。

允许插入的一端叫做队尾(rear);允许删除的另一端叫做队头(front)。队列又称先进先出(First in First Out,FIFO)表。

下面使用列表的 list. append()方法和 list. pop()方法实现堆栈和队列这两种数据结构。

```
#coding:utf-8
#例 4-20 使用列表实现堆栈和队列
#实现堆栈
#定义一个含有 5 个元素 (学号)的列表
student_list=['1001','1002','1002','1004','1005']
#输出初始化列表
print "初始化列表:",student_list
print "使用列表实现堆栈"
#调用 list.append()函数往列表尾部 (栈顶)添加元素
student_list.append('1006')
#输出添加尾部 (栈顶)元素后的列表
print "添加尾部 (栈顶)元素后的列表:",student_list
#调用 list.pop()函数删除列表尾部 (栈顶)的元素
student_list.pop()
#输出删除尾部 (栈顶)元素后的列表
print "删除尾部 (栈顶)元素后的列表:",student_list

#实现队列
print "使用列表实现队列"
#调用 list.append()函数往列表尾部 (队尾)添加元素
student_list.append('1006')
#输出添加尾部 (队尾)元素后的列表
print "添加尾部 (队尾)元素后的列表:",student_list
#调用 list.pop()函数删除列表头部 (队头)的元素
student_list.pop(0)
#输出删除头部 (队头)元素后的列表
print "删除头部 (队头)元素后的列表:",student_list
```

程序运行结果如下:

```
初始化列表：['1001', '1002', '1002', '1004', '1005']
使用列表实现堆栈
添加尾部(栈顶)元素后的列表：['1001', '1002', '1002', '1004', '1005', '1006']
删除尾部(栈顶)元素后的列表：['1001', '1002', '1002', '1004', '1005']
使用列表实现队列
添加尾部(队尾)元素后的列表：['1001', '1002', '1002', '1004', '1005', '1006']
删除头部(队头)元素后的列表：['1002', '1002', '1004', '1005', '1006']
```

该程序的初始化列表为['1001', '1002', '1002', '1004', '1005']。对于实现堆栈的数据结构，该列表我们可以称为堆栈。要想把元素'1006'放入栈顶(列表尾部)中，可以调用 list.append()函数，执行完该函数后就把元素'1006'放入栈顶中，此时，堆栈中的元素为['1001', '1002', '1002', '1004', '1005','1006']。出栈则可以调用 list.pop()函数。该函数没指定参数时默认是删除列表的尾部元素，执行完该函数后就把栈顶元素'1006'移出(删除)堆栈中，此时，堆栈中的元素为['1001', '1002', '1002', '1004', '1005']。对于实现队列的数据结构，该列表我们可以称为队列。要想把元素'1006'放入队尾(列表尾部)中，可以调用 list.append()函数，执行完该函数后就把元素'1006'放入队尾中，此时，队列中的元素为['1001', '1002', '1002', '1004', '1005','1006']。从队列中移出元素则可以调用 list.pop()函数。此时需要指定参数值 0，以移出队头(列表头部)元素，执行完该函数后就把队头元素'1001'移出队列中，此时，队列中的元素为['1002', '1002', '1004', '1005','1006']。

4.4　元　　组

元组和列表非常相似，它们都是序列类型，它们的元素都是用逗号分隔。元素类型可以是任意类型。但也有一些区别，从形式上看，列表是用方括号把元素括起来，而元组是用圆括号把元素括起来。从功能上看，列表是可变序列类型，而元组是不可变序列类型。

4.4.1　创建元组

创建一个元组，可以通过使用圆括号，并把圆括号里的每一个元素采用逗号进行分隔或者使用内建函数 tuple()来实现。元组的每一个元素可以是不同的数据类型。元组中的元素还可以是元组。

创建元组的一般格式如下：

```
tuple_name=(element1,element2,element3,…,elementN,)
```

注意，创建元组时，如果创建只含有一个元素的元组，元素后面的逗号不能省略，否则 Python 解析器就会认为该括号里面的内容是一个表达式，而不是元组的元素。创建含多个元素的元组时，最后一个元素后面的逗号可以省略。

```
#例 4-21　创建元组
#通过圆括号和逗号来创建元组
>>>num_tuple=(1,2,3,4,5)                    #元素都是整型
```

```
>>>string_tuple=('a','b','c')              #元素都是字符串
#元素类型有整型、浮点型、字符串类型和布尔型
>>>mix_tuple=(123,4.56,'abc',True)
#元素还可以是元组类型
>>>tuple_in_tuple=(num_tuple,string_tuple)
#通过内建函数 tuple()来创建元组
>>>another_num_tuple=tuple(range(5))
>>>print num_tuple
(1,2,3,4,5)
>>>print string_tuple
('a','b','c')
>>>print mix_tuple
(123,4.56,'abc',True)
>>>print tuple_in_tuple
((1,2,3,4,5),('a','b','c'))
>>>print another_num_tuple
(0,1,2,3,4)
```

4.4.2 访问元组

访问元组和访问列表非常类似,也是通过切片操作来实现的。返回的也是一个对象或者多个对象的集合。

元组的切片操作也遵循正负索引规则,也有开始索引和结束索引,如果这两个索引都没指定,默认也会分别指到序列的开始和结束位置。

```
#例 4-22  访问元组
>>>num_tuple=(1,2,3,4,5)                   #元素都是整型
>>>string_tuple=('a','b','c')              #元素都是字符串
#元素类型有整型、浮点型、字符串类型和布尔型
>>>mix_tuple=(123,4.56,'abc',True,(789,'789','efg'))
>>>num_tuple[1]
2
>>>num_tuple[1:3]
(2,3)
>>>num_tuple[1:]
(2,3,4,5)
>>>string_tuple[:3]
('a','b','c')
>>>string_tuple[-3:-1]
('b','c')
>>>mix_tuple[4]
(789,'789','efg')
>>>mix_tuple[-3:]
('abc',True,(789,'789','efg'))
```

访问二维元组可以通过如下方式：

```
tuple_name[index1][index2]
```

其中，index1 为二维元组的元素索引，index2 为二维元组中 index1 索引指向的元素中的元素索引。由于和访问二维列表非常相似，所以，这里就不举例说明了。

4.4.3　元组操作符

在这一小节，我们将结合例子来说明元组的操作符。

```
#例 4-23　元组操作符
>>>num_tuple=(12,-34,5.6,7.89e1)
>>>string_tuple=('I','want','to','learn','Python')
>>>mix_tuple=(1.2,'welcome',True,(3.4,'C'))

#1.连接操作符(+)
>>>num_tuple+string_tuple
(12,-34,5.6,78.9,'I','want','to','learn','Python')
>>>num_tuple+mix_tuple
(12,-34,5.6,78.9,1.2,'welcome',True,(3.4,'C'))
>>>string_tuple+mix_tuple
('I','want','to','learn','Python',1.2,'welcome',True,(3.4,'C'))

#2.重复操作符(*)
>>>num_tuple*2
[12,-34,5.6,78.9,12,-34,5.6,78.9]

#3.成员关系操作符(in、not in)
>>>7.89e1 in num_tuple
True
>>>'Python' in string_tuple
True
>>>'welcome' in mix_tuple
True
>>>'C' in mix_tuple
False
>>>'C' not in mix_tuple
True
>>>'C' in mix_tuple[3]
True
```

4.4.4　常用元组内建函数

表 4-9 列出了可以应用于元组的常用内建函数。

表 4-9 可以应用于元组的常用内建函数

函　　数	功能(注意,参数的内容并没有改变)
cmp(tuple1,tuple2)	比较两个元组中的元素,如果 tuple1 中的元素比 tuple2 中对应的元素大,则函数返回 1,如果小,则函数返回－1,如果 tuple1 和 tuple2 中所有的元素对应相等,则函数返回 0
len(tuple)	函数返回 tuple 元组的元素个数
max(tuple)	函数返回 tuple 元组中元素最大的值
min(tuple)	函数返回 tuple 元组中元素最小的值
sorted(tuple)	函数返回一个元组,该元组按从小到大的顺序排好
reversed(tuple)	函数返回已翻转的迭代器,翻转的意思是将第一个元素和最后一个元素对调,第二个元素和倒数第二个元素对调,以此类推
sum(tuple)	该函数只适用于元组中元素都是数字的时候,函数返回元组中所有数字的代数和
tuple(seq)	该函数返回的是一个元组,其元素来自于序列
type(obj)	函数返回的是 obj 对象的类型

下面将结合例子介绍其中的一些函数。

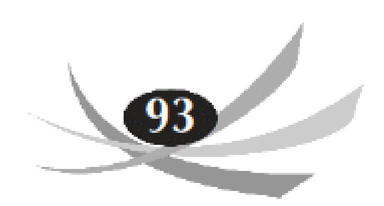
1. cmp()函数

使用 cmp()函数比较两个元组 tuple1 和 tuple2 时,其比较规则和 4.3.5 节中介绍的 cmp()函数比较规则一致,这里不再赘述。

```
#例 4-24  cmp()函数的使用
>>>num1_tuple=(1.23,-4,5.6,7.89e1)
>>>num2_tuple=(1.23,2.0,5.6,7.89e1)
>>>num3_tuple=(1.23,-4,5.6,7.89e1)
>>>string1_tuple=('I','want','to','learn','Python','and','Java')
>>>string2_tuple=('I','want','to','learn','Python')
#第一个元素相等,比较第二个元素,将-4转换成浮点数-4.0,小于 2.0,所以返回-1
>>>cmp(num1_tuple,num2_tuple)
-1
#两个元组中的所有元素都相等,所以返回 0
>>>cmp(num1_tuple,num3_tuple)
0
#类型不一致,数字类型小于字符串类型,所以返回-1
>>>cmp(num1_tuple,string1_tuple)
-1
#前 4个元素都分别相等,此时元组 string2_tuple 已到末尾,所以返回 1
>>>cmp(string1_tuple,string2_tuple)
1
```

2. sorted()函数和 reversed()函数

sorted()函数的作用是返回已排好序的元组,其元素是从小到大排序的。reversed()



函数的作用是返回已翻转的迭代器。

```
#coding:utf-8
#例 4-25  sorted()函数和 reversed()函数的使用
#定义一个混合类型元组
mix_tuple=('I',12,'want','to',3.4,'learn','Python')
#输出初始化元组
print '初始化元组为:',mix_tuple
#输出从小到大的顺序元组
print '排序后的元组为:',sorted(mix_tuple)
#使用 for 循环依次输出翻转后的迭代器中的元素
print '翻转后的迭代器中的元素为:',tuple(reversed(mix_tuple))
```

程序运行结果如下：

```
初始化元组为:('I', 12, 'want', 'to', 3.4, 'learn', 'Python')
排序后的元组为:(3.4, 12, 'I', 'Python', 'learn', 'to', 'want')
翻转后的迭代器中的元素为:('Python', 'learn', 3.4, 'to', 'want', 12, 'I')
```

4.5　本章小结

本章主要讲解了以下几个知识点：

(1) 序列类型。序列类型包括字符串、列表和元组类型。它们都有着相同的访问模式：通过指定一个下标位移量的方式可以访问到序列中的任何一个元素；通过切片的方式一次可以得到多个元素。下标位移量是从 0 开始到 N－1(N 序列中元素个数)结束或者从序列最后一个元素－1 开始到第一个元素－N 结束。

(2) 序列类型通用的操作符。包括成员关系操作符(in、not in)、连接操作符(＋)、重复操作符(*)、切片操作符([]、[:]、[: :])。

(3) 序列类型通用内建函数。包括类型转换函数(list()、str()、tuple())和可操作函数(len()、max()、min()等)。

(4) 字符串。字符串是一种不可变序列类型，可以通过在引号(包括单引号'、双引号"或三个引号''')间包含字符的方式或者通过内建函数 str()的方式创建字符串。灵活使用转义字符“\”。

(5) 列表。与字符串不同，列表是一种可变序列类型，可以通过使用方括号，并把方括号里的每一个元素采用逗号进行分隔或者使用内建函数 list()来创建列表。因为列表是可变序列类型，所以，可以改变列表的内容。改变列表的方式：添加元素，修改元素和删除元素。添加元素可以通过 list. append()函数或者 list. insert()函数实现；修改元素直接对需要修改的列表元素重新赋值即可；删除元素可以通过 list. remove()函数或者 list. pop()函数实现。对于堆栈和队列这两种数据结构，可以使用 list. append()方法和 list. pop()方法实现。

(6) 元组。元组和列表非常相似，它们都是序列类型，它们的元素都是用逗号分隔。

元素类型可以是任意类型。但也有一些区别，从形式上看，列表是用方括号把元素括起来，而元组是用圆括号把元素括起来。从功能上看，列表是可变序列类型，而元组是不可变序列类型。可以通过使用圆括号，并把圆括号里的每一个元素采用逗号进行分隔或者使用内建函数 tuple()来实现。

4.6 习　　题

一、解答题

1. 序列类型有哪些？它们的异同点是什么？它们又是如何访问其中的元素的？
2. 序列类型有哪些通用的操作符？

二、看程序写结果

1.

```
str='Python is cool!'
strjoin=''
for x in str:
    if x==' ':
        strjoin+='\n'
        continue
    strjoin+=x
print strjoin
```

2.

```
list=[]
i=0
for x in range(1,10):
    if x%2==1:
        list.append(x)
    else:
        list.insert(i,x)
        i+=1
print list
```

3.

```
list=[]
i=0
j=0
m=3
for x in range(1,10):
    if x%2==1:
        list.append(x)
    else:
        list.insert(i,x)
```

```
        i+=1
l=len(list)
while(j<m):
    t=list[l-1]
    k=l-1
    while(k>0):
        list[k]=list[k-1]
        k-=1
    j+=1
    list[k]=t
print list
```

4.

```
tuple=(1,2,3,4,5,6,7,8,9)
i=1
for x in range(1,9):
    if(i<0):
        x*=-1
        expr=tuple[x:i]
    else:
        expr=tuple[i:x]
        i*=-1
    print expr
```

三、上机练习

1. 编写程序输入一个字符串，分别输出按ASCII码顺序从小到大排好序的字符串、翻转的字符串和所有小写字母都变成大写字母的字符串。

2. 编写一个程序，输入两个日期，格式为yyyy-mm-dd，计算这两个日期间的天数并输出。

3. 编写程序，输入一个正整数，然后从个位开始一次输出每一位数字对应的英文字母。例如：输入1532，输出two three five one。

4. 用列表定义一个3×3的整型矩阵，求主对角线之和与副对角线之和的差值。

5. 将一个列表(其元素都是整数)中的元素先翻转，输出此时的列表，然后再将元素从小到大的顺序排序，最后输出排好序的列表，要求：不能使用list.reverse()和list.sort()这两个内建函数。

6. 有n个人围成一圈，顺序排号，从第1个开始报数(从1到3报数)，凡报到3的人退出圈子，编写程序，求最后留下的是原来第几号的那位。n由键盘输入。

7. 用元组定义一个4×4的整型矩阵，编写程序输出这个矩阵的所有鞍点，即该位置上的元素在该行上最大，在该列上最小。没有鞍点则输出“There is no saddle point”。

第 5 章 映射和集合类型

本章学习目标

- 掌握字典的创建、访问、更新
- 熟悉字典的常用内置函数
- 掌握集合的创建、访问、更新
- 熟悉集合的常用内置函数

在前面的章节中已经介绍了整型、浮点型、字符串、列表等数据类型。本章将介绍另外两种数据类型：映射(字典)和集合类型。

5.1 映射类型——字典

字典是 Python 语言中唯一的映射类型。这种映射类型由键(key)和值(value)组成(统称为“键值对”),一个键只能对应一个值,但多个键可以对应相同的值。字典对象是可变的数据类型,可以存储任意个键值对。字典中的值没有特定顺序,每个值都对应一个唯一的键,字典也被称作关联数组或哈希表。字典类型和序列类型的区别在于其存储和访问数据方式的不同。序列类型只用整型作为其索引,或者说只用整型作为其键。映射类型则可以用其他对象类型作为键。并且映射类型的键和其指向的值有一定的关联性,而序列类型则没有。正是由于映射类型的键可以“映射”到值,所以才称其为映射类型。

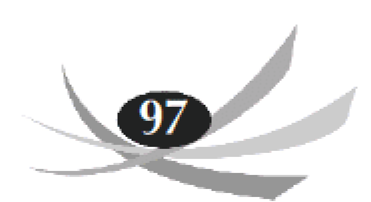
注意:字典的键必须是可哈希的对象,如字符串、整型、元组(元素不包含可变数据类型)都是可哈希的对象,都可以作为字典的键,而列表、字典是不可哈希的对象,所以不能用作字典的键。可以简单地把直接或间接不包含可变数据类型的对象看作可哈希的对象,当然,最妥当的方法还是通过 hash()函数来判断某个对象是否是可哈希的对象。

5.1.1 创建字典

字典由一系列的“键值对”组成,可以通过使用花括号,并把花括号里的每一个键值对采用逗号进行分隔,键值对中间用冒号隔开的方式来创建一个字典。

创建字典的一般格式如下:

```
dictionary_name={key1:value1,key2:value2,…,keyN:valueN}
```



其中,key1、key2、keyN 等表示字典的键,value1、value2、valueN 表示字典的键对应的值。

此外,还可以通过内建函数 dict()方法和 fromkeys()方法创建一个字典。dict()函数可以接收以(key,value)形式的列表或元组。使用 fromkeys()函数可以创建一个"默认"字典,字典中键对应的值都相同,如果没有指定值,默认为 None。

```
#例 5-1 创建字典
#1.通过普通方式来创建字典
#字典的键为数字,值为字符串
>>>student1_dict={1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
#字典的键为字符串,值也为字符串
>>>student2_dict={"1001":"xiaowang","1002":"xiaoli","1003":"xiaochen"}
>>>student1_dict
{1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
>>>student2_dict
{'1003': 'xiaochen', '1002': 'xiaoli', '1001': 'xiaowang'}

#2.通过内建函数 dict()来创建字典
#以(key,value)形式的列表
>>>student3_dict=dict([(1001,"xiaowang"),(1002,"xiaoli"),(1003,"xiaochen")])
#以(key,value)形式的元组
>>>student4_dict=dict(((1001,"xiaowang"),(1002,"xiaoli"),(1003,"xiaochen")))
>>>student3_dict
{1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
>>>student4_dict
{1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}

#3.通过内建函数 fromkeys()来创建字典
#指定 value 值为"person"
>>>student5_dict={}.fromkeys((1001,1002,1003),"person")
#不指定 value 值
>>>student6_dict={}.fromkeys((1001,1002,1003))
>>>student5_dict
{1001:"person",1002:"person",1003:"person"}
>>>student6_dict
{1001:None,1002:None,1003:None}
```

这个例子,先通过普通的方式创建了一个含有三个键值对的字典"student1_dict"。键为学生编号,类型为整型,值为编号对应的学生。接着又创建了字典"student2_dict",该字典的编号类型为字符串,输出时,可以发现输出的结果和所创建时给出的键值对顺序不一致。这是因为创建时给出的键值对顺序并不是字典的实际存储顺序,字典是根据每个键值对的键的 hashcode 值进行排序存储的。然后通过内建函数 dict(),分别以列表和元组的参数形式创建字典,最后再通过 fromkeys()函数创建两个字典。可以看到,当指

定 value 为“person”时，其键值对中的值都是“person”，当没指定 value 时，其键值对中的值都使用默认值“None”。

5.1.2 访问字典

访问字典中键值对的值可以通过方括号并指定相应的键的形式访问。需要注意的是，当指定一个字典中不存在的键时就会抛 KeyError 异常。遍历一个字典可以有以下几种方式：

(1) 通过指定键的方式遍历字典。在 Python 2.2 以前，需要使用 keys()函数获取字典的所有键。而在 Python 2.2 以后，可以直接遍历字典这个迭代器对象，每次返回的是字典的键，因此，可以通过 dictionary_name[key]的方式访问对应的值，从而可以遍历字典中所有的键值对。

(2) 通过内建函数 items()遍历字典。该函数返回的是一个由键值对组成的元组的列表。因此，可以遍历这个列表，从而遍历字典中所有的键值对。

(3) 通过内建函数 iteritems()遍历字典。该函数返回的是键值对的迭代器。因此，可以直接得到这个键值对，从而遍历字典中所有的键值对。

```
#coding:utf-8
#例 5-2  遍历字典
#定义含有三个键值对的字典
student_dict={1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
#1.1.通过指定键的方式遍历字典(Python 2.2 版本之前)
print "循环遍历 student_dict.keys():",student_dict.keys()
for key in student_dict.keys():
    print 'student_dict[%s]=' %key,student_dict[key]

#1.2.通过指定键的方式遍历字典(Python 2.2 版本之后)
print "循环遍历 student_dict:",student_dict
for key in student_dict:
    print 'student_dict[%s]=' %key,student_dict[key]

#2.通过内建函数 items()遍历字典
print "循环遍历 student_dict.items():",student_dict.items()
for (key,value) in student_dict.items():
    print 'student_dict[%s]=' %key,value

#3.通过内建函数 iteritems()遍历字典
print "循环遍历 student_dict.iteritems():",student_dict.iteritems()
for (key,value) in student_dict.iteritems():
    print 'student_dict[%s]=' %key,value
```

程序的运行结果如下：

```
循环遍历 student_dict.keys(): [1001, 1002, 1003]
```




```
student_dict[1001]=xiaowang
student_dict[1002]=xiaoli
student_dict[1003]=xiaochen
循环遍历 student_dict: {1001: 'xiaowang', 1002: 'xiaoli', 1003: 'xiaochen'}
student_dict[1001]=xiaowang
student_dict[1002]=xiaoli
student_dict[1003]=xiaochen
循环遍历 student_dict.items(): [(1001, 'xiaowang'), (1002, 'xiaoli'), (1003, 'xiaochen')]
student_dict[1001]=xiaowang
student_dict[1002]=xiaoli
student_dict[1003]=xiaochen
循环遍历 student_dict.iteritems():<dictionary-itemiterator object at 0x00000000027AFBD8
>
student_dict[1001]=xiaowang
student_dict[1002]=xiaoli
student_dict[1003]=xiaochen
```

这个程序首先定义了含有三个键值对(学生编号和对应的学生)的字典,然后三种方式遍历整个字典。第一种是通过指定键的方式遍历字典,这种方式又分两种情况:在Python版本2.2以前,可以通过key()函数获得字典的所有键组成的列表。如程序中输出student_dict.keys()就输出了[1001,1002,1003];另一种情况是在Python 2.2版本以后,可以直接遍历字典对象,每次返回的是字典中的键。因此,这两种情况都可以通过student_dict[key]的方式获取key键对应的值,从而遍历这个字典。第二种是通过内建函数items()遍历字典。这个函数返回的是一个由键值对组成的元组的列表,如输出student_dict.keys()就输出了[(1001,'xiaowang'),(1002,'xiaoli'),(1003,'xiaochen')]。因此,遍历这个列表就相当于遍历了字典。第三种是通过内建函数iteritems()遍历字典,这个函数返回的是一个键值对的迭代器,如输出student_dict.keys()就输出了<dictionary-itemiterator object at 0x00000000027AFBD8>。因此,遍历这个列表就相当于遍历了字典。

5.1.3　更新字典

字典是可变数据类型,即字典的长度和元素都是可以改变的。下面将介绍更新字典的方式:添加元素、修改元素和删除元素。

说明:本节讲的元素指定是键值对。

1. 添加元素

向字典添加一个元素可以通过赋值语句实现,该赋值语句的写法如下:

```
dictionary_name[key]=value
```

如果key在字典dictionary_name中不存在,则直接将元素(key,value)添加到字典中;如果key已存在,则value会覆盖原来字典中key对应的值,从而修改了字典中key对应的值(这种方式实现了修改元素的目的)。这种情况也相当于把元素(key,value)添

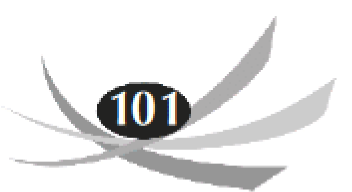
加到字典中，但同时也删除了原来字典中含 key 的元素，字典的元素个数没有增加。

```
#coding:utf-8
#例 5-3  使用赋值语句向字典添加一个元素
#定义一个含有三个元素 (学生编号:学生)的字典
student_dict={1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
#使用 len()函数获取 student_dict 字典中初始的个数,str()将整型转为字符串类型
print '目前有'+str(len(student_dict))+'个学生'
print '刚来了一个学生"xiaozhang",给他分配的学生编号为 1004'
#使用赋值语句向 student_dict 字典添加一个 (1004,"xiaozhang")的元素
student_dict[1004]="xiaozhang"
#再次输出此时 student_dict 字典的长度
print '现在有'+str(len(student_dict))+'个学生,他们分别是:'
#使用 for 循环遍历这个 student_dict 字典,分别输出这些元素
for key in student_dict:
    print 'student_dict[%s]=' %key,student_dict[key]
print '又来了一个学生"xiaoshui",给该学生分配一个该班里 (字典)已用过的学生编号 1004'
#使用赋值语句向 student_dict 字典添加一个 (1004,"xiaoshui")的元素
student_dict[1004]="xiaoshui"
#再次输出此时 student_dict 字典的长度
print '现在有'+str(len(student_dict))+'个学生,他们分别是:'
#使用 for 循环遍历这个 student_dict 字典,分别输出这些元素
for key in student_dict:
    print 'student_dict[%s]=' %key,student_dict[key]
```

程序运行结果如下：

```
目前有三个学生
刚来了一个学生"xiaozhang",给他分配的学生编号为 1004
现在有 4 个学生,他们分别是:
student_dict[1001]=xiaowang
student_dict[1002]=xiaoli
student_dict[1003]=xiaochen
student_dict[1004]=xiaozhang
又来了一个学生"xiaoshui",给该学生分配一个该班里 (字典)已用过的学生编号 1004
现在有 4 个学生,他们分别是:
student_dict[1001]=xiaowang
student_dict[1002]=xiaoli
student_dict[1003]=xiaochen
student_dict[1004]=xiaoshui
```

从程序运行结果可以看到，第一次使用赋值语句向字典添加元素(1004:"xiaozhang")后，遍历此时的字典，可以看到元素(1004:"xiaozhang")已被添加到字典中。第二次又使用赋值语句向字典添加元素(1004:"xiaoshui")后，遍历此时的字典，可以看到原来的元素(1004:"xiaozhang")已被新元素(1004:"xiaoshui")替换。或者说字典中键为"1004"



对应的 value 值已被修改成"xiaoshui"。字典的元素个数没有增加。这是因为添加的元素的键在字典中已存在。

此外，向字典中添加一个元素也可以通过 setdefault()内建函数实现。该函数的声明如下：

```
dictionary_name.setdefault(key[,default_value])
```

其中，dictionary_name 为字典名，参数 key 表示字典的键，参数 default_value 表示添加的字典元素默认的值。该参数为可选参数。如果不指定该参数的值，默认为 None。如果要添加的参数 key 在字典中已存在，那么该函数将返回原有的值。否则这个元素将被添加到字典中，并返回所添加元素的 value 值。下面通过一个例子来说明如何使用 setdefault()函数向字典添加一个元素。

```
#coding:utf-8
#例 5-4  使用 setdefault()函数向字典添加一个元素
#定义一个含有三个元素(学生编号:学生)的字典
student_dict={1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
#使用 len()函数获取 student_dict 字典中初始的个数,str()将整型转为字符串类型
print '目前有'+str(len(student_dict))+'个学生'
print '刚来了一个学生"xiaozhang",给他分配的学生编号为 1004'
#使用 setdefault()函数向 student_dict 字典添加一个(1004,"xiaozhang")的元素,并输
#出该函数的返回值
print student_dict.setdefault(1004,"xiaozhang")
#再次输出此时 student_dict 字典的长度
print '现在有'+str(len(student_dict))+'个学生,他们分别是:'
#使用 for 循环遍历这个 student_dict 字典,分别输出这些元素
for key in student_dict:
    print 'student_dict[%s]=' %key,student_dict[key]
print '又来了一个学生"xiaoshui",给该学生分配一个该班里(字典)已用过的学生编号 1004'
#使用 setdefault()函数向 student_dict 字典添加一个(1004,"xiaoshui")的元素,并输出
#该函数的返回值
print student_dict.setdefault(1004,"xiaoshui")
#再次输出此时 student_dict 字典的长度
print '现在有'+str(len(student_dict))+'个学生,他们分别是:'
#使用 for 循环遍历这个 student_dict 字典,分别输出这些元素
for key in student_dict:
    print 'student_dict[%s]=' %key,student_dict[key]
```

该程序运行的结果如下：

```
目前有三个学生
刚来了一个学生"xiaozhang",给他分配的学生编号为 1004
xiaozhang
现在有 4 个学生,他们分别是:
student_dict[1001]=xiaowang
student_dict[1002]=xiaoli
```

```
student_dict[1003]=xiaochen
student_dict[1004]=xiaozhang
又来了一个学生"xiaoshui",给该学生分配一个该班里(字典)已用过的学生编号 1004
xiaozhang
现在有 4 个学生,他们分别是:
student_dict[1001]=xiaowang
student_dict[1002]=xiaoli
student_dict[1003]=xiaochen
student_dict[1004]=xiaozhang
```

从程序运行结果可以看到,第一次使用 setdefault()函数向字典添加元素(1004:"xiaozhang")时,函数返回了"xiaozhang",说明元素添加成功。遍历此时的字典,可以看到元素(1004:"xiaozhang")确实已被添加到字典中。第二次又使用 setdefault()函数向字典添加元素(1004:"xiaoshui")时,函数返回了"xiaozhang",说明元素添加失败。遍历此时的字典,可以看到元素(1004:"xiaoshui")确实没有被添加到字典中。这是因为添加的元素的键在字典中已存在。

2. 修改元素

修改字典中的元素是通过赋值语句实现的,这在添加元素里面有提到。在赋值时指定的 key 值在字典中要存在。

```
#coding:utf-8
#例 5-5  修改字典
#定义一个含有 3 个元素(学生编号:学生)的字典
student_dict={1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
#输出初始化 student_dict 的字典
print '初始化的字典为:',student_dict
#使用赋值语句 student_dict 字典中 key 为 1001 对应的值改为"xiaochen",还有 key 为
#1003 对应的值改为"xiaowang",
student_dict[1001]="xiaochen"
student_dict[1003]="xiaowang"
#输出修改后 student_dict 的字典
print '修改后的字典为:',student_dict
```

该程序运行的结果如下:

```
初始化的字典为:{1001: 'xiaowang', 1002: 'xiaoli', 1003: 'xiaochen'}
修改后的字典为:{1001: 'xiaochen', 1002: 'xiaoli', 1003: 'xiaowang'}
```

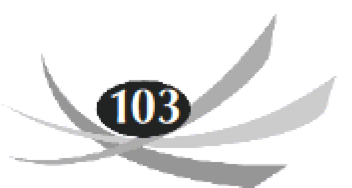
3. 删除元素

删除字典中的元素可以通过 del()函数、pop()函数或 del 语句实现,下面将分别介绍这几种方式。

(1) 通过 del()函数删除字典中的元素

该函数的语法格式如下:

```
del(dictionary_name[key])
```



其中,key表示所要删除的元素的键(key),且字典中存在这个键。

(2) 通过pop()函数删除字典中的元素

该函数的语法格式如下:

```
dictionary_name.pop(key[,default_value])
```

其中,key表示所要删除的元素的键(key),如果字典中存在这个key,则函数返回key所对应的值,否则返回default_value。

(3) 通过del语句删除字典中的元素

该语句的语法格式如下:

```
del dictionary_name[key]
```

其中,key表示所要删除的元素的键(key),del语句和del()函数在功能上都是一样的,在形式上只是del语句没有括号而已。

注意:使用del()函数或del语句删除不存在的元素时会抛KeyError异常,而使用pop()函数则不会。

下面通过一个例子来说明这三种删除字典中元素的方式的用法。

```
#coding:utf-8
#例5-6　删除字典中的元素
#定义一个含有三个元素(学生编号:学生)的字典
student_dict={1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
#输出初始化student_dict的字典
print '初始化的字典为:',student_dict

#1.使用del()函数删除字典中的键为1001的元素
del(student_dict[1001])
#输出此时的student_dict字典
print '使用del()函数删除字典中的键为1001的元素后的字典为:',student_dict

#2.1.使用pop()函数删除字典中的键为1001的元素(不存在)
print student_dict.pop(1001,"不存在键为1001的元素")
#2.2.使用pop()函数删除字典中的键为1002的元素(存在)
print student_dict.pop(1002,"不存在键为1002的元素")
#输出此时的student_dict字典
print '使用pop()函数删除字典中的键为1002的元素后的字典为:',student_dict

#3.使用del语句删除字典中的键为1003的元素
del student_dict[1003]
#输出此时的student_dict字典
print '使用del语句删除字典中键为1003的元素后的字典为:',student_dict
```

程序运行结果如下:

```
初始化的字典为:{1001: 'xiaowang', 1002: 'xiaoli', 1003: 'xiaochen'}
```

```
使用 del()函数删除字典中键为 1001 的元素后的字典为：{1002: 'xiaoli', 1003: 'xiaochen'}
不存在键为 1001 的元素
xiaoli
使用 pop()函数删除字典中键为 1002 的元素后的字典为：{1003: 'xiaochen'}
使用 del 语句删除字典中键为 1003 的元素后的字典为：{}
```

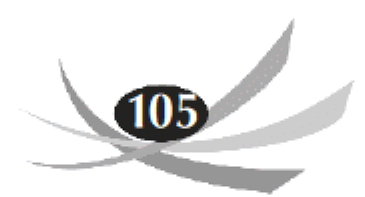
从程序运行结果可以看到，第一次使用 del()函数删除字典中键为“1001”的元素后，此时的字典有两个元素，分别为(1002,'xiaoli')和(1003,'xiaochen')。第二次使用 pop()函数删除字典中键为“1001”的元素，由于此元素在字典中已不存在，所以该函数返回默认值“不存在键为 1001 的元素”。接下来再次使用 pop()函数删除字典中键为“1002”的元素。由于此元素在字典中存在，所有该函数返回 1002 对应的值“xiaoli”。此时的字典只有一个元素(1003,'xiaochen')。最后使用 del 语句删除字典中键为“1003”的元素，可以看到此时的字典已变成空字典{}。

5.1.4 字典操作符

适合于字典的操作符只有两个，一个是键查找操作符([])，这个操作符和序列类型中的单一元素切片操作符很相似。通过这个操作符，既可以获取字典中指定的元素，也可以给字典中指定的元素赋值。这在前面的例子中已有所体现。另外一个操作符是成员关系操作符(in、not in)，这个操作符和序列类型中成员关系操作符一样。

下面通过一个例子来理解这两个操作符。

```
#coding:utf-8
#例 5-7  字典操作符的使用
#定义一个含有三个元素（学生编号:学生）的字典
student_dict={1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
print '初始化的字典为:',student_dict
op=input("请输入对应的数字选择相应的操作（0:删除元素 1:修改元素）")
if op==0:
    key=input("请输入要删除的元素对应的键（-1表示停止删除）:")
    while key!=-1:
        #判断所输入的 key 在字典中是否存在
        if key in student_dict:
            #key 在字典中存在，则删除该元素
            del student_dict[key]
            print "删除%s所对应的元素后的字典为:" %key,student_dict
            #若字典长度为零，则跳出循环
            if len(student_dict)==0:
                print "字典已为空，无法继续删除元素"
                break
        else:
            print "您输入的 key 值（%s）在字典中不存在" %key
        key=input("请输入要删除的元素对应的键（-1表示停止删除）:")
else:
```




```
key=input("请输入要修改的元素对应的键(-1表示停止修改):")
while key!=-1:
    #判断所输入的key在字典中是否存在
    if key in student_dict:
    #key在字典中存在,则先输入value值,再将value值赋给要修改的元素
        value=raw_input("请输入要修改的值")
        student_dict[key]=value
        print "将%s赋给%s所对应的元素后的字典为:" %(value,key),student_dict
    else:
        print "您输入的key值(%s)在字典中不存在" %key
    key=input("请输入要修改的元素对应的键(-1表示停止修改):")
```

若输入1,程序的运行结果如下(包括必要的输入):

```
初始化的字典为:{1001: 'xiaowang', 1002: 'xiaoli', 1003: 'xiaochen'}
请输入对应的数字选择相应的操作(0:删除元素 1:修改元素)1
请输入要修改的元素对应的键(-1表示停止修改):1001
请输入要修改的值xiaochen
将xiaochen赋给1001所对应的元素后的字典为:{1001: 'xiaochen', 1002: 'xiaoli', 1003: '
xiaochen'}
请输入要修改的元素对应的键(-1表示停止修改):1003
请输入要修改的值xiaowang
将xiaowang赋给1003所对应的元素后的字典为:{1001: 'xiaochen', 1002: 'xiaoli', 1003: '
xiaowang'}
请输入要修改的元素对应的键(-1表示停止修改):1004
您输入的key值(1004)在字典中不存在
请输入要修改的元素对应的键(-1表示停止修改):-1
```

该程序先输入1,表示要执行修改操作。然后输入要修改的元素对应的键(key)1001,程序使用成员关系操作符(in)来判断输入的key在字典中是否存在,如果存在,输入要修改的值"xiaochen",再通过赋值语句修改元素的值,此时键"1001"对应的值已变为"xiaochen"。依次输入1003、"xiaowang"时,键"1003"对应的值则变为"xiaowang"。再次输入要修改的元素对应的键"1004",由于该键在字典中不存在,所以输出"您输入的key值(1004)在字典中不存在",然后再输入-1来退出程序。

若输入0,程序的运行结果如下(包括必要的输入):

```
初始化的字典为:{1001: 'xiaowang', 1002: 'xiaoli', 1003: 'xiaochen'}
请输入对应的数字选择相应的操作(0:删除元素 1:修改元素)0
请输入要删除的元素对应的键(-1表示停止删除): 1001
删除1001所对应的元素后的字典为:{1002: 'xiaoli', 1003: 'xiaochen'}
请输入要删除的元素对应的键(-1表示停止删除):1001
您输入的key值(1001)在字典中不存在
请输入要删除的元素对应的键(-1表示停止删除):1002
删除1002所对应的元素后的字典为:{1003: 'xiaochen'}
请输入要删除的元素对应的键(-1表示停止删除):1003
```

```
删除 1003 所对应的元素后的字典为：{}
字典已为空,无法继续删除元素
```

该程序先输入 0，表示要执行删除操作。然后输入要删除的元素对应的键(key) 1001，"1001"在字典中存在，直接删除该元素，此时的字典为：{1002：'xiaoli'，1003：'xiaochen'}。当再次输入 1001 时，由于字典中已不存在"1001"的元素，所以输出"您输入的 key 值(1001)在字典中不存在"。依次输入 1002、1003。程序将删除这两个键对应的元素，删除后的字典已为空，自动退出程序。

5.1.5 常用字典内建函数

1. 适用于字典的常用内建函数

表 5-1 列出了可以应用于字典的常用内建函数。

表 5-1 可以应用于字典的常用内建函数

函　数	功能(注意，参数的内容并没有改变)
cmp(dict1,dict2)	比较两个字典的大小，首先比较的是字典的元素个数，如果 dict1 中的元素个数比 dict2 中的元素个数多，则函数返回 1，否则函数返回 −1，如果元素个数相等，则比较键，最后再比较值
len(dict)	函数返回 dict 字典的元素个数
hash(obj)	该函数用于判断 obj 对象是否可以作为字典的键，如果可以则返回这个对象的哈希值(一个整数)，否则会抛 TypeError 异常，如字典和列表就不能作为字典的键
dict([container])	如果提供了容器类(container)，则创建的字典的元素来自于该容器类，否则创建的是一个空字典
type(obj)	函数返回的是 obj 对象的类型

cmp()函数用于字典的比较，使用的情况非常少，所以就不展开来介绍了。这里主要介绍 hash()函数。

hash()函数可以用于判断某个对象是否可以作为字典的键，将一个对象作为参数传给 hash()函数，如果这个对象是可哈希的，则会返回这个对象的哈希值，也只有可哈希的对象才可以作为字典的键。如果某个不可哈希的对象作为参数传给 hash()函数，则会抛 TypeError 异常，该对象就不能作为字典的键。

```
#coding:utf-8
#例 5-8  hash()函数的使用

#参数为整型
>>>hash(123)
123                                    #整型 123 是可哈希,可以作为字典的键
>>>key_int_dict={123:'python'}
>>>key_int_dict
{123:'python'}
```

```
#参数为字符串
>>>hash('123')
1911471187                            #字符串'123'是可哈希,可以作为字典的键
>>>key_str_dict={'123':'python'}
>>>key_str_dict
{'123':'python'}

#参数为列表
>>>hash([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
TypeError: unhashable type: 'list'    #抛 TypeError 异常,提示列表是不可哈希的类型
>>>key_list_dict={[1,2,3]:'python'}
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
TypeError: unhashable type: 'list'    #所以使用列表作为键创建字典时就会出错(抛
                                      #TypeError 异常)

#参数为元组(没有可变数据类型)
>>>hash((1,2,3))
-378539185                            #字符串'123'是可哈希,可以作为字典的键
>>>key_tuple_dict={(1,2,3):'python'}
>>>key_tuple_dict                     #元组(1,2,3)是可哈希,可以作为字典的键
{(1,2,3):'python'}

#参数为元组(有可变数据类型)
>>>hash((1,2,3,[1,2,3]))
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
TypeError: unhashable type: 'list'    #因为元组中含有可变数据类型列表,而列表是不
                                      #可哈希的类型,所以同样会抛 TypeError 异常

#参数为字典
>>>hash({123:'python'})
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
TypeError: unhashable type: 'dict'    #抛 TypeError 异常,提示字典是不可哈希的类型
```

从这个例子可以看到整型、字符串、不包含可变数据类型的元组都是可哈希的对象,都可以作为字典的键,而列表、字典都是不可哈希的对象,因此,都不能作为字典的键。

2. 字典对象的常用内建函数

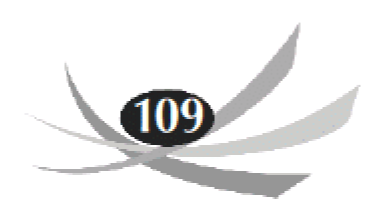
表5-2列出了字典对象的常用内建函数。其中的一些函数已在前面介绍过。

表5-2 字典对象的常用内建函数

函　　数	功　　能
dict. clear()	删除字典中的所有元素
dict. copy()	返回字典的一个副本
dict. fromkeys(seq, value =None)	创建并返回一个新字典,以序列 seq 中元素作为字典的键,value 为字典所有键对应的初始值,默认值为 None
dict. get(key, default = None)	返回指定键 key 对应的值,如果字典中不存在这个键 key,则返回 default 值,默认为值 None
dict. has_key(key)	如果键 key 在字典中,则返回 true,否则返回 false
dict. items()	返回以字典中的键和值组成的元组的一个列表
dict. keys()	返回一个包含字典中所有键的列表
dict. setdefault (key, default=None)	该函数用于向字典添加一个元素,若指定的 key 在字典中存在,则返回该元素的值,添加元素失败,若指定的 key 在字典中不存在,则向字典添加一个元素(key,default),default 默认值为 None
dict. pop(key[, default])	如果字典中存在这个键 key,则删除并返回 key 对应的值,如果字典中不存在这个键 key,则返回 default 值(若没有给出 default 值,则会抛 KeyError 异常)
dict. update(dict2)	把字典 dict2 的键值对添加到字典 dict 中
dict. values()	返回一个包含字典中所有值的列表

下面举例介绍其中的几个函数。

```
#coding:utf-8
#例 5-9  字典对象的常用内建函数
#定义一个含有三个元素(学生编号:学生)的字典
student1_dict={1001:"xiaowang",1002:"xiaoli",1003:"xiaochen"}
#再定义一个只含有一个元素(学生编号:学生)的字典
student2_dict={1004:"xiaohe"}
#输出初始化字典
print "初始化字典 student1_dict:",student1_dict
#使用 get()函数返回键'1001'对应的值
print "输出字典中键'1001'对应的值:没有则返回'no such key'",student1_dict.get(1001,'no
such key')
#使用 pop()函数删除键为'1001'的元素
student1_dict.pop(1001)
print "删除键为'1001'的元素后的字典:",student1_dict
#使用 has_key()函数判断此时字典是否有键'1001'
print student1_dict.has_key('1001')
print "输出字典中键'1001'对应的值:没有则返回'no such key'",student1_dict.get(1001,'no
such key')
#使用 copy()函数返回字典 student2_dict 的副本,然后再使用 update()函数将该字典副本
#添加到字典 student1_dict 中
```




```
student1_dict.update(student2_dict.copy())
print "修改后的字典 student1_dict:",student1_dict
#使用 values()函数返回字典 student1_dict 的所有值
print "字典 student1_dict 的所有值",student1_dict.values()
```

程序运行结果如下：

```
初始化字典 student1_dict: {1001: 'xiaowang', 1002: 'xiaoli', 1003: 'xiaochen'}
输出字典中键'1001'对应的值,没有则返回'no such key:' xiaowang
删除键为'1001'的元素后的字典: {1002: 'xiaoli', 1003: 'xiaochen'}
false
输出字典中键'1001'对应的值,没有则返回'no such key:' no such key
修改后的字典 student1_dict: {1002: 'xiaoli', 1003: 'xiaochen', 1004: 'xiaohe'}
字典 student1_dict 的所有值: ['xiaoli', 'xiaochen', 'xiaohe']
```

该程序首先使用 get()函数返回字典 student1_dict 中键为“1001”对应的值“xiaowang”，然后使用 pop()函数删除键为“1001”的元素，接着使用 has_key()函数判断此时的字典 student1_dict 是否还存在键为“1001”的元素，输出“False”，说明删除成功。此时再使用 get()函数返回字典 student1_dict 中键为“1001”对应的值就得到 get()函数设置的默认值“no such key”。接下来使用 copy()函数返回字典 student2_dict 的副本，然后再使用 update()函数将该字典副本添加到字典 student1_dict 中。从输出中可以看到字典 student2_dict 已成功添加到字典 student1_dict 中，最后使用 values()函数返回此时字典 student1_dict 的所有值。

5.2　集合类型

Python 中的集合类型是从数学中的集合概念引进来的。在 Python 中，集合也是一种无序不重复的元素集。集合中的元素要求是可哈希的对象。集合有两种不同的类型，可变集合(set)和不可变集合(frozenset)。对于可变集合，则可以对集合中的元素进行添加和删除。而不可变集合则不可以添加和删除元素。集合的基本用途包括成员关系测试和重复条目消除。集合对象也支持合(union)、交(intersection)、差(difference)以及对称差(symmetric difference)等数学操作。

在详细介绍 Python 集合对象之前，先通过表 5-3 来回忆数学中关于集合的运算符号以及认识 Python 中对应的操作符。

表 5-3　集合操作符和成员关系符

数学符号	Python 符号	说　明
$\in$	in	是……的成员
$\notin$	not in	不是……的成员
$=$	==	等价
$\neq$	!=	不等价

续表

数 学 符 号	Python 符号	说　　明
⊂	<	是……的子集
⊆	<=	是……的真子集
⊃	>	是……的超集
⊇	>=	是……的真超集
∩	&	交集
∪	|	并集
－或/	－	相对补集
Δ	^	对称差分

5.2.1　创建集合

由于集合有两种不同的类型,因此,创建的方式也不同。如果创建一个可变集合,可以通过使用花括号,并把花括号里的每一个元素采用逗号进行分隔的方式或者通过内建函数 set()来实现。而要创建一个不可变的集合,只能通过内建函数 frozenset()实现。

创建可变集合的一般格式如下:

```
set_name={element1,element2,element3,…,elementN}
```

其中,element1,element2,elementN 等表示集合的元素。

注意:使用 set()或 frozenset()函数创建集合时,参数是一个列表,如果创建的是一个单字符多元素的集合,则参数可以是一个字符串。

```
#例 5-10　创建集合
#1.创建可变集合
#1.1.通过普通方式创建
>>>student1_set={"xiaowang","xiaoli","xiaochen"}
>>>student1_set
set(['xiaochen', 'xiaoli', 'xiaowang'])

#1.2.通过 set()函数的方式创建
>>>student2_set=set(["xiaowang","xiaoli","xiaochen","xiaochen"])
>>>student2_set
set(['xiaochen', 'xiaoli', 'xiaowang'])

#2.创建不可变集合
>>>student3_set=frozenset(["xiaowang","xiaoli","xiaochen"])
>>>student3_set
frozenset(["xiaowang","xiaoli","xiaochen"])
```

读者可能已经注意到,在使用 set()函数创建 student2_set 集合时,参数是一个有 4

个元素的列表，且后两个元素相同。由于集合中的元素不能重复，所以，创建出来的集合student2_set只有三个元素('xiaochen', 'xiaoli', 'xiaowang')。

5.2.2 访问集合

由于集合中的元素是无序的，并且也没有像字典中“键值对”的对应关系，所以，无法获取某个指定的元素，只能遍历整个集合来访问其中的所有元素。

```
#coding:utf-8
#例 5-11  遍历集合
#定义含有三个元素的集合
student_set={"xiaowang","xiaoli","xiaochen"}
#通过 for 循环遍历这个集合
print 'student_set 集合含有以下元素:'
for element in student_set:
    print element
```

程序的运行结果如下：

```
student_set 集合含有以下元素:
xiaochen
xiaoli
xiaowang
```

5.2.3 更新集合(可变集合)

这里说的更新集合只是更新可变集合，因为只有可变集合中的元素才能够修改，而不可变集合则不能修改。并且这里的更新也只有向集合中添加元素和从集合中删除元素，而没有修改元素，因为无法获取某个指定的元素。下面将介绍这两种更新集合的方式。

1. 添加元素

向集合添加一个元素可以通过内建函数set.add()实现，而一次性添加多个元素可以通过set.update()函数实现，这个函数其实是将一个集合中的所有元素添加到set集合中，当然，还要去掉其中重复的元素。

```
##coding:utf-8
#例 5-12  向集合中添加元素
#定义一个含有两个元素的集合
student1_set={"xiaoli","xiaochen"}
#再定义一个含有三个元素的集合
student2_set={"xiaowang","xiaoshui","xiaode"}
#输出初始化集合 student1_set
print "初始化集合 student1_set 为:",student1_set
#使用 add()函数向集合 student1_set 中添加元素"xiaowang"
student1_set.add("xiaowang")
print "使用 add()函数添加元素'xiaowang'后的集合 student1_set 为:",student1_set
```

```
#使用 update()函数一次性向集合 student1_set 中添加多个元素
student1_set.update(student2_set)
print "使用 update()函数一次性添加多个元素后的集合 student1_set 为:",student1_set
```

程序运行结果如下：

```
初始化集合 student1_set 为: set(['xiaochen', 'xiaoli'])
使用 add () 函 数 添 加 元 素 ' xiaowang ' 后 的 集 合 student1 _ set 为: set ([' xiaochen ',
'xiaoli', 'xiaowang'])
使用 update () 函 数 一 次 性 添 加 多 个 元 素 后 的 集 合 student1 _ set 为: set ([' xiaode ',
'xiaochen', 'xiaoli', 'xiaowang', 'xiaoshui'])
```

2. 删除元素

删除集合中的元素可以通过 remove()函数、pop()函数、discard()函数或 clear()函数实现，下面将分别介绍这几个函数。

(1) 通过 remove()函数删除集合中的元素

该函数的语法格式如下：

```
set_name.remove(obj)
```

其中，obj 表示所要删除的元素(对象)。如果 obj 不是集合中的元素，将会抛 KeyError 异常。

(2) 通过 pop()函数删除集合中的元素

该函数的语法格式如下：

```
set_name.pop()
```

该函数删除集合中的第一个元素并将该元素返回。

(3) 通过 discard()函数删除集合中的元素

该语句的语法格式如下：

```
set_name.discard(obj)
```

其中，obj 表示所要删除的元素。这个函数与 remove()函数的区别就是：即使 obj 不是集合中的元素，程序也不会抛 KeyError 异常。

(4) 通过 clear()函数删除集合中的元素

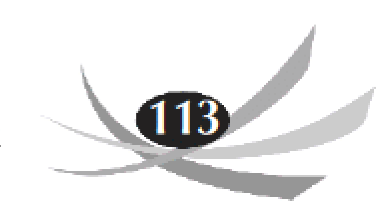
该语句的语法格式如下：

```
set_name.clear()
```

该函数将删除集合中的所有元素。

下面通过一个例子来说明这 4 种删除集合中元素的方式的用法。

```
#coding:utf-8
#例 5-13  从集合中删除元素
#定义一个含有两个元素的集合
student_set={"xiaowang","xiaoli","xiaochen"}
```




```
#输出初始化集合 student_set
print "初始化集合 student_set 为:",student_set
#1. 使用 remove()函数删除集合 student_set 中的"xiaowang"
student_set.remove("xiaowang")
#输出此时的集合 student1_set
print "使用 remove()函数删除'xiaowang'后的集合 student_set 为:",student_set
#2. 使用 pop()函数随机删除集合 student_set 中的一个元素
obj=student_set.pop()
print "使用 pop()函数删除集合 student_set 中的第一个元素后的集合 student_set 为:",
student_set
#3. 使用 discard()函数删除集合 student_set 中剩下的那个元素
if obj=='xiaoli':
    #删除"xiaochen"
    student_set.remove("xiaochen")
    print "使用 discard()函数删除'xiaochen'后的集合 student_set 为:",student_set
else:
    #删除"xiaoli"
    student_set.remove("xiaoli")
    print "使用 discard()函数删除'xiaoli'后的集合 student_set 为:",student_set
#对 student_set 重新赋值
student_set={"xiaowang","xiaozhang","xiaohe"}
print "重新赋值后的集合 student_set 为:",student_set
#4. 使用 clear()函数删除集合 student_set 中所有的元素
student_set.clear()
print "使用 clear()函数后的集合 student_set 为:",student_set
```

程序运行结果如下：

```
初始化集合 student_set 为: set(['xiaochen', 'xiaoli', 'xiaowang'])
使用 remove()函数删除'xiaowang'后的集合 student_set 为: set(['xiaochen', 'xiaoli'])
使用 pop()函数删除集合 student_set 中的第一个元素后的集合 student_set 为: set
(['xiaoli'])
使用 discard()函数删除'xiaoli'后的集合 student_set 为: set([])
重新赋值后的集合 student_set 为: set(['xiaowang', 'xiaohe', 'xiaozhang'])
使用 clear()函数后的集合 student_set 为: set([])
```

5.2.4　集合操作符

从表 5-3 中可以看出集合操作符非常丰富。其中，成员关系操作符和序列类型或字典类型中的成员关系操作符是一样的，这里就不介绍了。此外，等价、不等价、子集、真子集、超集和真超集等操作符是通过关系运算符实现的，所以这里也不对这些操作符进行介绍了。下面将介绍交集、并集、相对补集和对称差分集操作符。

1. 并集(|)

集合 s 和集合 t 进行并操作，即 s|t，将会产生一个新的集合，这个集合中的元素要么

属于集合 s，要么属于集合 t 或者既属于集合 s 又属于集合 t。

2. 交集(&)

集合 s 和集合 t 进行交操作，即 s&t，也会产生一个新的集合，这个集合中的元素既属于集合 s 又属于集合 t。

3. 相对补集(-)

集合 s 和集合 t 进行相对补操作，即 s-t，同样会产生一个新的集合，这个集合中的元素属于集合 s，并且不属于集合 t。

4. 对称差分集(^)

集合 s 和集合 t 进行对称差分操作，即 s^t，所产生的新集合，其中的元素只能仅属于集合 s 或者仅属于集合 t。

下面通过一个例子来理解这 4 个操作符。

```
#coding:utf-8
#例 5-14  集合操作符的使用
#分别定义两个含有三个元素的集合
student1_set={"xiaowang","xiaoli","xiaochen"}
student2_set={"xiaowang","xiaohe","xiaozhang"}
#输出初始化集合
print '初始化集合 student1_set 为:',student1_set
print '初始化集合 student2_set 为:',student2_set
#1.两集合进行并操作
print 'student1_set|student2_set:',student1_set|student2_set

#2.两集合进行交操作
print 'student1_set&student2_set:',student1_set&student2_set

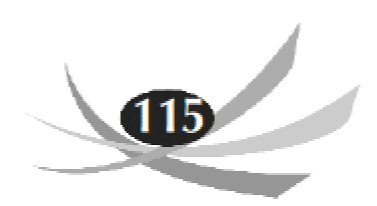
#3.两集合进行相对补操作
print 'student1_set-student2_set:',student1_set-student2_set

#4.两集合进行对称差分操作
print 'student1_set^student2_set:',student1_set^student2_set
```

程序的运行结果如下：

```
初始化集合 student1_set 为: set(['xiaochen', 'xiaoli', 'xiaowang'])
初始化集合 student2_set 为: set(['xiaowang', 'xiaohe', 'xiaozhang'])
student1_set|student2_set: set(['xiaozhang', 'xiaochen', 'xiaoli', 'xiaowang', 'xiaohe'])
student1_set&student2_set: set(['xiaowang'])
student1_set-student2_set: set(['xiaochen', 'xiaoli'])
student2_set-student1_set: set(['xiaohe', 'xiaozhang'])
student1_set^student2_set: set(['xiaozhang', 'xiaohe', 'xiaoli', 'xiaochen'])
```

该程序中，定义的两个集合分别有三个元素，但只有一个元素是相同的。因此，在两



集合进行并操作时得到的集合有5个元素;进行交操作时,得到的集合只有它们共有的元素"xiaowang";进行 student1_set-student2_set 的相对补操作时,得到的集合的元素有两个,它们只属于集合 student1_set,且不属于集合 student2_set,而进行 student2_set-student1_set 的相对补操作时,得到的集合的元素也是两个,它们只属于集合 student2_set,且不属于集合 student1_set;进行对称差分操作时,得到的集合有4个元素(除了它们共用的那个元素)。

5.2.5 常用集合内建函数

1. 适用于集合的常用内建函数

表5-4列出了可以应用于集合的三个内建函数。

表5-4　可以应用于集合的常用内建函数

函　　数	功能(注意,参数的内容并没有改变)
len(set)	该函数返回集合 set 的元素个数
set([obj])	该函数返回一个可变集合,其中的元素来自 obj 对象,且 obj 是可迭代的
frozenset([obj])	该函数返回一个不可变集合,其中的元素来自 obj 对象,且 obj 是可迭代的

2. 集合对象的常用内建函数

表5-5列出了所有集合对象的常用内建函数。

表5-5　集合对象的常用内建函数

函　　数	等价操作符	说　　明
s.issubset(t)	s<=t	如果 s 是 t 的子集,则返回 true,否则返回 false
s.issuperset(t)	s>=t	如果 s 是 t 的超集,则返回 true,否则返回 false
s.union(t)	s|t	函数返回集合 s 和 t 的并集
s.intersection(t)	s&t	函数返回集合 s 和 t 的交集
s.difference(t)	s-t	函数返回集合 s 和 t 的相对补集
s.symmetric_difference(t)	s^t	函数返回集合 s 和 t 的对等差分集
s.copy()		函数返回集合 s 的副本

对于例5-14,下面通过函数的方式求两个集合的并、交、相对补、对称等分这4个操作:

```
#coding:utf-8
#例 5-15　集合对象的常用内建函数
#分别定义两个含有三个元素的集合
student1_set={"xiaowang","xiaoli","xiaochen"}
student2_set={"xiaowang","xiaohe","xiaozhang"}
#输出初始化集合
print '初始化集合 student1_set 为:',student1_set
```

```
print '初始化集合 student2_set 为:',student2_set
#1.通过 union()函数实现两集合的并操作
print 'student1_set.union(student2_set):',student1_set.union(student2_set)

#2.通过 intersection()函数实现两集合的交操作
print 'student1_set.intersection(student2_set):',student1_set.intersection(student2_
set)

#3.通过 difference()函数实现两集合的相对补操作
print 'student1_set.difference(student2_set):',student1_set.difference(student2_set)
print 'student2_set.difference(student1_set):',student2_set.difference(student1_set)

#4.通过 symmetric_difference()函数实现两集合的交操作
print 'student1_set.symmetric_difference(student2_set):',student1_set.symmetric_
difference(student2_set)
```

程序运行结果如下：

```
初始化集合 student1_set 为: set(['xiaochen', 'xiaoli', 'xiaowang'])
初始化集合 student2_set 为: set(['xiaowang', 'xiaohe', 'xiaozhang'])
student1_set.union(student2_set): set(['xiaozhang', 'xiaochen', 'xiaoli',
'xiaowang', 'xiaohe'])
student1_set.intersection(student2_set): set(['xiaowang'])
student1_set.difference(student2_set): set(['xiaochen', 'xiaoli'])
student2_set.difference(student1_set): set(['xiaohe', 'xiaozhang'])
student1_set.symmetric_difference(student2_set): set(['xiaozhang', 'xiaohe', 'xiaoli', '
xiaochen'])
```

从例 5-14 和 5-15 可以看到通过函数和通过操作符实现的集合间的并集、交集、相对补集、对称差分集是一样的。

表 5-6 列出了所有集合对象的常用内建函数。其中，有大部分函数在前面已经介绍过。

表 5-6 集合对象的常用内建函数

函　　数	等价操作符	说　　明
s.update(t)	s|=t	将 t 中的元素添加到 s 中
s.intersection_update(t)	s&=t	更新集合 s，使集合 s 中的元素仅属于集合 s 和 t 的共同元素
s.difference_update(t)	s-=t	更新集合 s，使集合 s 中的元素仅属于集合 s 而不属于集合 t
s.symmetric_difference_update(t)	s^=t	更新集合 s，使集合 s 中的元素仅属于集合 s 或仅属于集合 t
s.add(obj)		将 obj 对象添加到集合 s 中
s.remove(obj)		从集合 s 中删除 obj 对象，若 obj 不在集合中将抛 KeyError 异常

续表

函　　数	等价操作符	说　　明
s. discard(obj)		从集合 s 中删除 obj 对象，即使 obj 不在集合中，执行这条语句也不会抛 KeyError 异常
s. pop()		该函数删除集合中的任意一个元素并将该元素返回
s. clear()		该函数删除集合中的所有元素

表中的 s. intersection_update(t)、s. difference_update(t)和 s. symmet ric_difference_update(t)函数和前面介绍的 s. intersection(t)、s. difference(t)和 s. symmetric_difference(t)函数非常相似，只是前者更新了集合 s，且没有返回值，或者说返回“None”。而后者没有更新集合 s，仅返回一个新的集合。

5.3　本章小结

本章主要讲解了以下几个知识点：

(1) 字典。字典是 Python 语言中唯一的映射类型。这种映射类型由键(key)和值(value)组成(统称为“键值对”)，字典对象是可变的数据类型，它的值没有特定顺序。字典类型和序列类型的区别在于其存储和访问数据方式的不同。序列类型只用整型作为其索引，或者说只用整型作为其键。映射类型则可以用其他对象类型作为键。

(2) 字典的创建、访问和更新。字典的创建可以通过使用花括号，并采用逗号将每个键值对隔开，键值对中间用冒号分隔的方式或使用内建函数 dict()方法和 formkeys()方法实现。字典的访问可以通过指定键的方式、遍历内建函数 items()或 iteritems()的返回对象的方式实现。而字典的更新又包括添加元素、修改元素和删除元素。添加元素可以通过赋值语句或内建函数 setdefault()实现；修改元素也是通过赋值语句实现；删除元素可以通过 del()函数、pop()函数或 del 语句实现。

(3) 集合。集合是一种无序不重复的元素集。集合中的元素要求是可哈希的对象。集合有两种不同的类型，可变集合和不可变集合。对于可变集合，则可以对集合中的元素进行添加和删除。而不可变集合则不行。

(4) 集合的创建、访问和更新。由于集合有两种不同的类型，因此，创建的方式也不同。如果创建一个可变集合，可以通过使用花括号，并采用逗号将花括号内元素分隔的方式或者通过内建函数 set()来实现。而要创建一个不可变的集合，只能通过内建函数 frozenset()实现。由于集合中的元素是无序的，并且也没有像字典中“键值对”的对应关系，所以，无法获取某个指定的元素，只能通过 for 循环遍历整个集合来访问其中的所有元素。而集合的更新包括添加元素和删除元素。添加元素可以通过内建函数 add()或 update()实现；删除元素可以通过 remove()函数、pop()函数、discard()函数或 clear()函数实现。

5.4 习　　题

一、解答题

1. 字典中的元素有什么特点？列表可以作为字典中元素的键吗？为什么？
2. 集合有哪些类型？可以修改集合中指定元素的值吗？为什么？

二、看程序写结果

1.

```
student_score_dict={1001:84}
student_score_dict[1002]=90
score=0
print student_score_dict.setdefault(1002,80)
print student_score_dict.setdefault(1003,96)
for key in student_score_dict:
    score+=student_score_dict[key]
print "the student's' average socre is",score/3.0
```

2.

```
num_set={1}
i=9
for x in range(1,10):
    if x%2==1:
        num_set.add(i)
        i-=1
    else:
        i-=2
    if i<0:
        break
print num_set
```

3.

```
num1_set={1}
num2_set={1}
i=9
j=9
for x in range(1,10):
    if x%2==1:
        num1_set.add(i)
        i-=1
        j-=3
    else:
        num2_set.add(j)
        i-=2
        j-=1
```

```
        if i<0 or j<0:
            break
print 'num1_set|num2_set:',num1_set|num2_set
print 'num1_set&num2_set:',num1_set&num2_set
print 'num1_set-num2_set:',num1_set-num2_set
print 'num1_set^num2_set:',num1_set^num2_set
```

三、上机练习

1. 依次把整型、浮点型、布尔型、字符串、列表、元组、字典、集合作为字典的键，看看哪些类型可以，哪些类型不行？为什么？

2. 定义一个含有三个元素的字典，分别用三种方式删除其中的一个元素，然后再输出该字典。

3. 定义一个含有三个元素的可变集合，分别用三种方式删除其中的一个元素，然后再输出该集合。

4. 定义两个集合 num1_set＝{1,2,3,4,5,6}和 num1_set＝{2,4,6}，使用函数和操作符的方式判断这两个集合，哪个是另外一个的子集，哪个是另外一个的超集，并且，同样使用函数和操作符的方式求出它们的交集、并集、相对补集和对称差分集。

5. 编写一个程序，首先通过键盘输入的方式将用户信息保存到一个字典中，要求循环依次提示输入用户的账号(若输入－1 则结束输入)、密码、住址、联系方式。然后依次提示输入用户名和密码，以实现登录功能。若用户名在字典中不存在则提示“用户名不存在！”，然后继续输入用户名。若密码不正确则提示“密码不正确！”，然后继续输入密码。若用户名和密码在字典中存在且相互对应。则提示“登录成功！”。如图 5-1 所示。

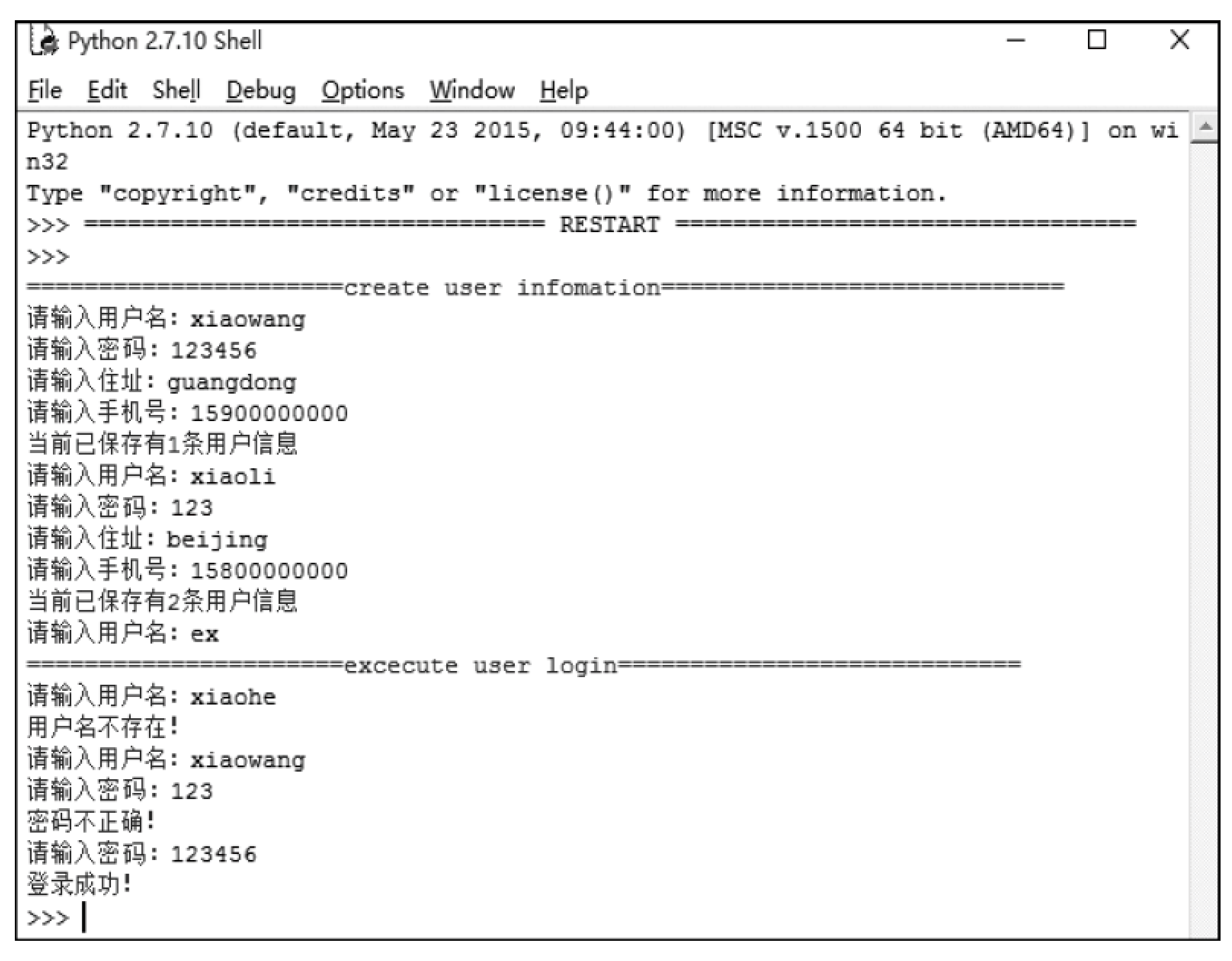


图 5-1　创建用户信息并实现登录功能

第 6 章　函　　数

本章学习目标

- 熟练掌握函数
- 理解函数的分类
- 理解函数参数的分类并能够灵活使用
- 掌握函数的嵌套调用
- 掌握函数的递归调用
- 掌握变量的作用域

通过前面章节的介绍，相信读者已经对 Python 的数据类型、运算符和控制结构有了一定的认识。但是，怎样减少程序的代码量，同时提高程序的执行效率和可维护性呢？通过本章的学习，相信读者会找到答案。

6.1　概　　述

Python 程序是由包（package）、模块（module）、函数（function）组成的。其中，包是由一系列模块组成的集合，而模块是处理某一类问题的函数和类的集合。它们的关系如图 6-1 所示。

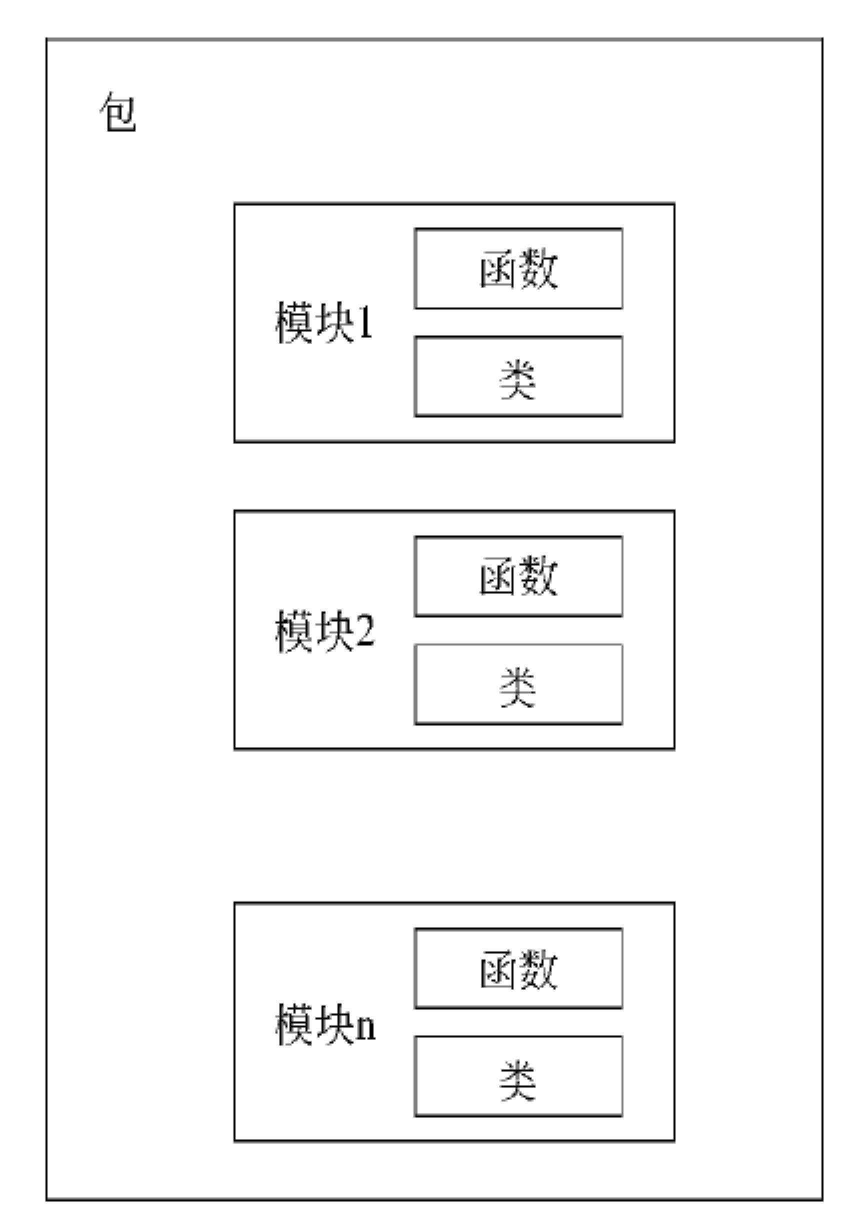


图 6-1　包、模块、类和函数的关系

本章将介绍函数，类、模块和包将分别在第 7 章、第 8 章中介绍。

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。函数能减少程序的代码量，节约了存储空间，同时也提高应用的模块性以及代码的重复利用率和程序的可维护性。

下面先举一个简单的函数调用的例子。

```
#coding:utf-8
#例 6-1　简单函数的定义和调用
#定义 print_start 函数
def print_start():
    print "*****************************"
#定义 print_message 函数
```

```
def print_message(who):
    print "          hello "+who+"!          "
#对 print_start 函数的调用
print_start()
#对 print_message 函数的调用
print_message("Mr.Right")
#再次调用 print_start 函数
print_start()
```

程序的运行结果如下：

```
***************************************
          hello Mr.Right!
***************************************
```

这个程序首先定义了 print_start 函数,这个函数是一个无参函数,只是用来输出一排“*”。接下来又定义 print_message 函数,该函数有一个传入参数,根据该参数的值来输出相应的欢迎信息。然后分别调用这两个函数。

从用户使用的角度来看,函数有两种。

(1) 内建函数。例如前面介绍的用于输入的 input 函数和用于判断数据类型的 type 函数等,这些函数都是系统提供的,用户不必自己定义,而是可以直接使用它们。

(2) 用户自定义函数。由用户根据需要自己定义的用于解决用户特定问题的函数。如例 6-1 的 print_start 函数和 print_message 函数。

从函数的形式看,函数分为两类。

(1) 无参函数。如例 6-1 的 print_start 函数就是无参函数。调用无参函数时不需要传入数据。无参函数一般用来执行一组指定的操作。例如,例 6-1 的 print_start 函数的作用是输出一排的星号。无参函数可以带回或不带回函数值(用 return 语句返回的数值)。但大部分的无参函数都不带回函数值。

(2) 有参函数。如例 6-1 的 print_message 函数就是有参函数。调用有参函数时需要传入相关的数据(除非参数都定义成默认参数)。并且,一般情况下,调用有参函数都会返回函数值。例 6-1 的 print_message 函数只是为了简单起见就没有返回函数值。

6.2　函数的定义

6.2.1　无参函数的定义

定义无参函数的一般形式为：

```
def 函数名():
    "文档字符串"
    函数体
```

其中,def 是保留字,用于定义函数,接着是函数名,其命名规则和变量标识符命名规

则一样。再然后是一对圆括号和一个冒号。第二行是一个可选的函数功能说明的字符串,但建议加上这个函数说明字符串,这样,当别人调用你写的函数时,可以通过函数名.__doc__获取该函数的功能说明,即可以清晰地知道该函数有什么用途,其中的__doc__是函数的其中一个属性,属性将在本章后面小节中介绍。第三行起就是函数体了。函数体中可能包含 return 语句,用于返回函数值。

例 6-1 中的 print_start 函数就是无参函数。

6.2.2 有参函数的定义

定义有参函数的一般形式为:

```
def 函数名(形式参数列表):
    "文档字符串"
    函数体
```

定义有参函数和定义无参函数基本上是一样的,但有一个明显的区别,就是函数名后面的括号有形式参数表列。而且函数体中可以引用这些参数。对于有参函数,一般都会返回函数值。

例如下面这个例子。

```
#coding:utf-8
#例 6-2  有参函数的定义
#定义 max 函数
def max(a,b):
    "求 a 和 b 两者中较大者"
    if(a>b):
        c=a
    else:
        c=b
    return c
```

这是一个求 a 和 b 两者中较大者的函数,max 是函数名。括号中有两个形式参数 a 和 b。在函数调用时,把实际参数的值传递给被调函数中形式参数 a 和 b。函数的第二行指定该函数的功能是“求 a 和 b 两者中较大者”。在函数体中,把 a 和 b 中的较大者赋给 c,然后用 return 语句返回这个 c 值,这样,在函数的调用处就可以使用该返回值了。

6.2.3 空函数

在程序设计中有时会用到空函数,它的定义形式为:

```
def 函数名():
    "文档字符串"
    pass
```

定义空函数时,函数体只有一条 pass 语句,pass 语句其实就是空语句,表示什么都不执行。

例如：

```
#coding:utf-8
#例 6-3　空函数的定义
def max():
    "求参数中较大者"
    pass
```

调用该函数时，将起不到任何的作用。在程序的指定位置写上“max()”表示“此处要调用一个函数”，而此时该函数不起作用，等以后扩展函数功能时再补充。

在程序设计的第一阶段往往先确定主要功能的函数，其他一些次要功能的函数则在以后需要时再陆续补上。在编写程序的开始阶段，可以在将来需要扩充功能的地方写上一个空函数，以后等编写好该函数时再代替它。这样做，使得程序的结构清楚，可读性好，以后扩展新功能方便。这对于刚开始设计一个复杂的程序是比较有用的。

6.3　函数参数和函数返回值

本节主要是针对有参函数。在定义有参函数时函数名后面括号中的变量名就称为“形式参数”(简称“形参”)，在调用有参函数时，函数名后面括号中的参数称为“实际参数”(简称“实参”)。

形式参数又分为位置参数(以正确的位置顺序传入函数的参数)、关键字参数(以顺序或不按顺序传入，但带有参数列表中曾定义过的关键字)、默认参数(函数调用时不一定要指定的参数)和可变长度参数(每次函数调用时的参数数目允许不相同)。实际上，关键字参数是对实际参数来说的。

下面先介绍函数的参数传递，然后再分别介绍形式参数的类型，最后再简单介绍函数的返回值。

6.3.1　参数传递

在 Python 语言中，实参向形参的参数传递都是采用引用传递的方式。在函数调用时，实参传递引用给形参，使得实参和形参都指向相同的对象。但对于不同的对象(这里的对象分为可变对象 mutable object 和不可变对象 immutable object。可变对象如列表、字典等，不可变对象如整型、浮点型、字符串和元组等)，又会表现出不同的结果。当实参变量所指向的对象是可变对象时，改变形参所指向的对象实际上也改变了实参所指向的对象。这时 Python 中的引用传递和 C++ 中的引用传递是一样的。当实参变量所指向的对象是不可变对象时，改变形参变量，如重新赋值，但由于形参变量所指向的对象不可变，所以形参变量将指向新的对象，而实参变量还是指向之前的对象。从而表现出“改变形参变量，实参变量并没有改变”。这是 Python 中的引用传递和 C++ 中的引用传递不一样的地方。实际上，严格来说是一样的，不一样是因为在 Python 中，有些对象是不能更改的。

下面通过例子分别介绍这两种情况。

例 6-4 中的实参变量指向不可变对象(整型)。

```
#coding:utf-8
#例 6-4  实参变量指向不可变对象
def change(a,b):
    "改变 a 和 b 两者的值"
    #使用内置的 id 函数求出变量的内存地址
    print "形参 a 的内存地址为:",id(a)
    print "形参 b 的内存地址为:",id(b)
    a,b=0,1
    print "对形参 a 重新赋值后的内存地址为:",id(a)
    print "对形参 b 重新赋值后的内存地址为:",id(b)

def main():
    "调用 change 函数"
    #实参变量指向不可变对象
    x,y=3,5
    print "实参 x 的内存地址为:",id(x)
    print "实参 y 的内存地址为:",id(y)
    print "调用函数前:x=%d,y=%d" % (x,y)
    change(x,y)
    print "调用函数后:x=%d,y=%d" % (x,y)
main()
```

程序的运行结果如下：

```
实参 x 的内存地址为: 40198200
实参 y 的内存地址为: 40198152
调用函数前:x=3,y=5
形参 a 的内存地址为: 40198200
形参 b 的内存地址为: 40198152
对形参 a 重新赋值后的内存地址为: 40198272
对形参 b 重新赋值后的内存地址为: 40198248
调用函数后:x=3,y=5
```

程序首先定义了一个改变 a 和 b 两者的值的 change 函数，a、b 为该函数的形式参数，然后再定义一个 main 函数，在这个函数中，变量 x、y 的值分别被赋值为 3 和 5(即分别指向对象 3 和对象 5)，先使用内置的 id 函数求出此时实参变量 x、y 的内存地址分别为 40198200 和 40198152(每次运行程序时内存地址都不一样)。再输出调用 change 函数前变量 x、y 的值。接着调用 change 函数，此时 change 函数括号内的 x、y 为实参。在调用函数时，实参把引用传递给形参。使得实参变量 x 和形参变量 a 指向相同的对象 3，实参变量 y 和形参变量 b 指向相同的对象 5，如图 6-2 所示。

在 change 函数中，首先输出此时形参变量 a、b 的内存地址分别为 40198200 和 40198152。这和实参变量的内存地址一样，即验证了函数调用时实参变量和形参变量都指向相同的对象。然后对 a、b 重新赋值，由于 a、b 指向的对象是不可变对象。所以，将为 0 和 1 分配内存单元，同时使 a、b 分别指向这两个对象。接着再输出形参变量 a、b 重新

赋值后的内存地址分别为40198272和40198248，这和赋值前的内存地址不一样，说明对形参a、b重新赋值时不是改变a、b指向的对象，而是使a、b分别指向新的对象0和对象1，实参x、y还是指向原来的对象，如图6-3所示。

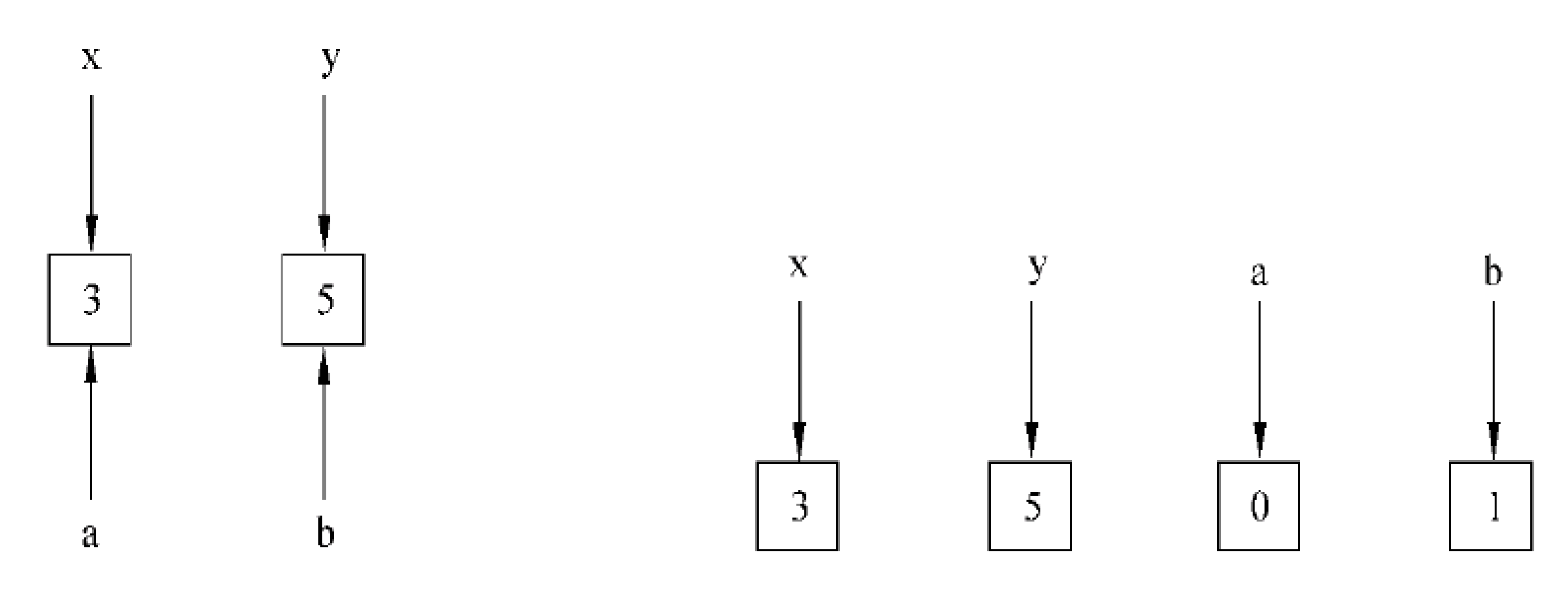


图6-2　实参与形参指向相同的变量　　**图6-3　实参和形参指向不同的变量**

调用结束后，在main函数输出此时变量x、y的值。由于在change函数中，x、y指向的对象一直没变(也不能改变)，所以它们的值还是3和5。

接下来分析另一种情况。

例6-5中的实参变量指向可变对象(列表)。

```
#coding:utf-8
#例 6-5  实参变量指向可变对象
def change(a,b):
    "改变 a 和 b 两者的值"
    #使用内置的 id 函数求出变量的内存地址
    print "形参 a 的内存地址为:",id(a)
    print "形参 b 的内存地址为:",id(b)
    a[1],b[1]=4,5
    print "对形参 a 重新赋值后的内存地址为:",id(a)
    print "对形参 b 重新赋值后的内存地址为:",id(b)

def main():
    "调用 change 函数"
    #实参变量指向不可变对象
    x,y=[1,2],[1,3]
    print "实参 x 的内存地址为:",id(x)
    print "实参 y 的内存地址为:",id(y)
    print "调用函数前 x 为",x
    print "调用函数前 y 为",y
    change(x,y)
    print "调用函数后 x 为",x
    print "调用函数后 y 为",y
main()
```

程序的运行结果如下：

```
实参 x 的内存地址为: 48764936
```

```
实参 y的内存地址为：41833288
调用函数前 x为 [1, 2]
调用函数前 y为 [1, 3]
形参 a的内存地址为：48764936
形参 b的内存地址为：41833288
修改形参 a后的内存地址为：48764936
修改形参 b后的内存地址为：41833288
调用函数后 x为 [1, 4]
调用函数后 y为 [1, 5]
```

该程序和例 6-4 有所不同。在 main 函数中，变量 x、y 的值分别被赋值为[1, 2]和[1, 3]，输出此时实参变量 x、y 的内存地址分别为 48764936 和 41833288。接着调用 change 函数，在调用函数时，实参把引用传递给形参。使得实参变量 x 和形参变量 a 指向相同的对象[1, 2]，实参变量 y 和形参变量 b 指向相同的对象[1, 3]，如图 6-4 所示。

在 change 函数中，首先输出此时形参变量 a、b 的内存地址分别为 48764936 和 41833288。这和实参变量的内存地址一样，同样验证了函数调用时实参变量和形参变量都指向相同的对象。然后使 a、b 所指向的列表对象中的第二个元素分别赋值为 4 和 5，接着再输出形参变量 a、b 所指向的对象被修改后的内存地址分别为 48764936 和 41833288，和修改前的内存地址一样，即形参 a、b 的指向并没有发生改变，只是改变了 a、b 指向的对象，实参 x、y 还是指向之前的对象，但指向的对象的内容发生了变化，如图 6-5 所示。

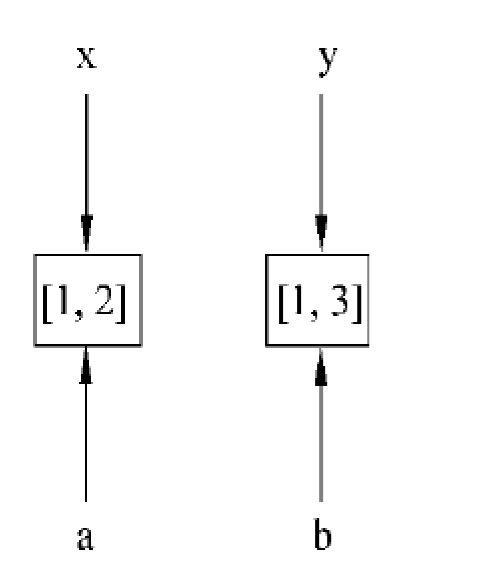


图 6-4　例 6-5 中实参与形参的指向 1

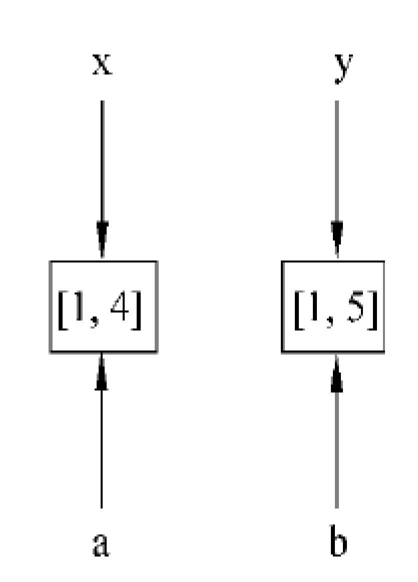


图 6-5　例 6-5 中实参与形参的指向 2

调用结束后，在 main 函数输出此时变量 x、y 的值。由于在 change 函数中，x、y 指向的对象的内容通过形参 a、b 被改变，所以它们的值是被修改后的[1, 4]和[1, 5]。

6.3.2　位置参数

位置参数无须特殊声明，例如，例 6-5 的 change 函数中的 a、b 都是位置参数。如果函数声明的参数是位置参数，则在函数调用时所给出的实参顺序应和函数定义中的位置参数顺序一致，否则程序将得不到正确的结果，如果参数个数不对应会抛 TypeError 异常。

```
#coding:utf-8
#例 6-6　位置参数
```

```
def message(who,programming_language):
    '打印谁想学习什么编程语言'
    print who+' want to learn '+programming_language+'!'
#实参顺序和函数定义中的位置参数顺序一致
message('Mr.right','Python')
#实参顺序和函数定义中的位置参数顺序不一致
message('Python','Mr.Right')
#实参个数和函数定义中的位置参数个数不同
message('Mr.right','Python')
```

程序的运行结果如下：

```
Mr.Right want to learn Python!
Python want to learn Mr.Right!

Traceback (most recent call last):
  File "C:/Python27/6.6.py", line 9, in<module>
    message('Mr.right')
TypeError: message() takes exactly 2 arguments (1 given)
```

从程序运行结果可以看出，第二个函数调用 message('Python','Mr. Right')，实参顺序和函数定义中的位置参数顺序不一致，输出 Python want to learn Mr. Right!，这并不是想要的结果。如果实参个数和函数定义中的位置参数个数不同。第二个函数调用 message('Mr. Right')，实参个数和函数定义中的位置参数个数不同，程序抛 TypeError 异常，提示需要两个参数，但只给了一个参数。

6.3.3　默认参数

如果函数声明的参数是默认参数，则在函数调用时可以不指定该参数的值，这时该参数将取函数声明时的默认值。设置默认参数是为了提高程序的灵活性和便捷性。因为在某些应用中，有些参数在大多数情况下是一样的，只有少数情况是不同的，把该参数设置成默认参数，其值取大多数情况下的值。这样，对于大多数情况下的函数调用都可以不要指定该参数的值，而取默认值。只有在少数情况下的函数调用，才根据需要指定默认参数的值。从而提高了程序的灵活性和便捷性。

使用默认参数和在电脑上安装软件的过程类似。软件的安装包在每一步都设置了默认参数，这样我们在安装软件时可以一直单击“下一步”按钮直到安装完成，这既方便，又节省时间。我们在安装软件时确实大部分都选择默认安装，当然了，我们也可以根据自己需要修改安装时设置的默认参数。

含默认参数的函数声明如下：

```
def 函数名 ([posargs,] defarg1=defval1,defarg2=defval2,...)
    "文档字符串"
    函数体
```

其中，posargs 是位置参数，可以没有位置参数，但如果存在，则它必须要在所有的默

认参数前面，defarg1，defarg2 是默认参数名，defval1，defval2 是默认值。

下面通过例 6-7 来理解默认参数。

```
# coding:utf-8
# 例 6-7  默认参数
def distance(second,gravity=9.8):
    '刚开始做自由落体运动的物体在给定的时间内运动的位移'
    m_distance=1/2.0 * gravity * second * second
    return m_distance

# 不指定默认参数的值
result=distance(5)
print '指定位置参数 second 的值为 5,不指定默认参数,而使用默认值 9.8 的结果为:',result
# 指定默认参数的值为 10
result=distance(5,10)
print '指定位置参数 second 和默认参数 gravity 的值分别为 5 和 10 的结果为:',result
```

程序的运行结果如下：

```
指定位置参数 second 的值为 5,不指定默认参数 gravity,而使用默认值 9.8 的结果为:122.5
指定位置参数 second 和默认参数 gravity 的值分别为 5 和 10 的结果为:125.0
```

该程序的第一个函数调用 distance(5)，仅指定位置参数 second(运动的时间)的值为 5，不指定默认参数，这时默认参数就取函数声明时的默认值 9.8。运行的结果为 24.5，即一个刚开始做自由落体运动的物体，在重力加速度为 9.8km/s^2，运动时间为 5 秒所移动的位移为 122.5 米。第二个函数调用 distance(5,10)，指定位置参数 second 的值为 5，默认参数的值为 10，由于指定了默认参数的值，那么函数声明的默认值 9.8 将被指定的参数值 10 所覆盖。运行结果为 125.0，即一个刚开始做自由落体运动的物体，在重力加速度为 10km/s^2，运动时间为 5 秒所移动的位移为 125.0 米。

6.3.4 关键字参数

在 6.3.2 节中，我们使用的参数是位置参数，即调用该函数给其传值时，是根据实参的先后顺序来给这些位置参数一一赋值的。不过有时候如果我们需要定义的函数有大量的参数，这时仅依赖于顺序传值是比较容易出错的，此时我们可以利用 Python 的关键字参数。使用关键字参数不需要对函数进行任何修改，只需要在调用函数时显式地为参数赋值即可(可以不依赖特定顺序)。例如，对例 6-7，函数不变，只是函数调用有些不同。

注意：关键字参数要放在其他参数的后面。

```
# coding:utf-8
# 例 6-8  关键字参数
def distance(second,gravity=9.8):
    '刚开始做自由落体运动的物体在给定的时间内运动的位移'
    m_distance=1/2.0 * gravity * second * second
    return m_distance
```

```
#使用关键字参数,给定的实参顺序和函数声明的形参顺序一致
result=distance(second=5,gravity=10)
print '先指定关键字参数 second 的值为 5,再指定关键字参数 gravity 的值为 10 的结果为:',
result
#使用关键字参数,给定的实参顺序和函数声明的形参顺序不一致
result=distance(gravity=10,second=5)
print '先指定关键字参数 gravity 的值为 10,再指定关键字参数 second 的值为 5 的结果为:',
result
#使用关键字参数和默认参数
result=distance(second=5)
print '指定关键字参数 second 的值为 5,使用默认参数值 9.8 的结果为:',result
```

程序的运行结果如下:

```
先指定关键字参数 second 的值为 5,再指定关键字参数 gravity 的值为 10 的结果为: 125.0
先指定关键字参数 gravity 的值为 10,再指定关键字参数 second 的值为 5 的结果为: 125.0
指定关键字参数 second 的值为 5,使用默认参数值 9.8 的结果为: 122.5
```

该程序的第一个函数调用 distance(second＝5,gravity＝10),先指定关键字参数 gravity 的值为 10,再指定关键字参数 second 的值为 5,此时给定的实参顺序和函数声明的形参顺序一致。第二个函数调用 distance(gravity＝10,second＝5),先指定关键字参数 gravity 的值为 10,再指定关键字参数 second 的值为 5,此时给定的实参顺序和函数声明的形参顺序不一致。但这两个函数调用的结果都是 125.0,这是因为关键字参数并不依赖于函数声明时的参数顺序。第三个函数调用 distance(second＝5),仅指定关键字参数 second 的值为 5,使用默认参数值 9.8,这个函数调用的函数返回值为 122.5。

6.3.5 可变长度参数

前面介绍的参数都是固定的,即都是一对一的关系,但有时可能会需要用到可变长度参数。在这种情况下,参数的数目是不确定的。把参数设置成可变参数,在函数调用时,指定的位置可以有多个实参,从而提高了程序的灵活性。由于函数调用有关键字参数和非关键字参数调用的方式,相应的,Python 有两个形式的可变长度参数。一种是把非关键字的可变长度参数存储在一个元组中,另一种是把关键字的可变长度参数存储在一个字典中。下面分别介绍这两个可变长度参数。

1. 非关键字的可变长度参数

当函数调用时,所有的位置参数和默认参数从实参列表中获得对应值之后,剩下的非关键字参数将按顺序插入一个元组中。

含非关键字的可变长度参数的函数声明如下:

```
def 函数名([posargs,defarg=defval] * vargs_tuple)
    "文档字符串"
    函数体
```

其中,vargs_tuple 为非关键字的可变长度参数,它在位置参数和默认参数之后,并以元组的方式保存了匹配完前面的位置参数和默认参数后剩下的所有参数,如果没有剩下的参数,则该元组为空。在 Python 语言中,通过星号 * 来声明非关键字的可变长度参数,而在 C 语言中是通过三个点...来声明可变参数的。

下面通过一个例子来理解非关键字的可变长度参数。

```
#coding:utf-8
#例 6-9  非关键字的可变长度参数
def NonkeywordVarArgs(posarg,defarg='defVal',*varargs):
    '演示非关键字的可变长度参数'
    print '位置参数 posarg:',posarg
    print '默认参数 defarg:',defarg
    i=1
    for eachvar in varargs:
        print '非关键字可变长度参数 vararg'+str(i)+':',eachvar
        i+=1
    print '-----------------------------------------------------'

#指定位置参数和默认参数,不指定非关键字的可变长度参数
NonkeywordVarArgs(123,'abc')
#指定位置参数,不指定默认参数和非关键字的可变长度参数
NonkeywordVarArgs(123)
#指定位置参数和默认参数,同时指定一个非关键字的可变长度参数
NonkeywordVarArgs(123,'abc',456)
#指定位置参数和默认参数,同时指定两个非关键字的可变长度参数
NonkeywordVarArgs(123,'abc',456,'def')
#指定位置参数和默认参数,同时指定三个非关键字的可变长度参数
NonkeywordVarArgs(123,'abc',456,'def',[1,'a'])
```

程序的运行结果如下:

```
位置参数 posarg: 123
默认参数 defarg: abc
-----------------------------------------------------
位置参数 posarg: 123
默认参数 defarg: defval
-----------------------------------------------------
位置参数 posarg: 123
默认参数 defarg: abc
非关键字可变长度参数 vararg1: 456
-----------------------------------------------------
位置参数 posarg: 123
默认参数 defarg: abc
非关键字可变长度参数 vararg1: 456
非关键字可变长度参数 vararg2: def
```

```
--------------------------------------------------
位置参数 posarg: 123
默认参数 defarg: abc
非关键字可变长度参数 vararg1: 456
非关键字可变长度参数 vararg2: def
非关键字可变长度参数 vararg3: [1, 'a']
--------------------------------------------------
```

该程序的第一个函数调用 NonkeywordVarArgs(123,'abc')和第二个函数调用 NonkeywordVarArgs(123)都不指定非关键字的可变长度参数,所以可变长度参数 varargs 是一个空的元组,没有输出。后面的三个函数调用分别指定了1、2和3个非关键字的可变长度参数。相应的,元组 varargs 也分别存有1、2和3个参数值,遍历这个元组,输出其中的参数值。

2. 关键字可变长度参数

当函数调用时,所有的位置参数和默认参数从实参列表中获得对应值之后,剩下的关键字参数将插入一个字典中。

含关键字可变长度参数的函数声明如下:

```
def 函数名([posargs,defarg=defval,*vargs_tuple,] **vargs_dist)
    "文档字符串"
    函数体
```

其中,vargs_dist 为关键字可变长度参数,它以字典的方式保存了匹配完前面的位置参数,默认参数和非关键字的可变长度参数后,剩下的所有关键字参数。下面通过一个例子来理解关键字可变长度参数。

```
#coding:utf-8
#例 6-10  关键字可变长度参数
def keywordVarArgs(posarg,defarg='defVal',*varargs,**kwvarargs):
    '演示关键字可变长度参数'
    print '位置参数 posarg:',posarg
    print '默认参数 defarg:',defarg
    i=1
    for eachvar in varargs:
        print '非关键字可变长度参数 vararg'+str(i)+':',eachvar
        i+=1
    i=1
    for key in kwvarargs:
        print '关键字可变长度参数 '+key+':',kwvarargs[key]
        i+=1
    print '--------------------------------------------------'

#指定位置参数、默认参数和一个非关键字的可变长度参数,不指定关键字可变长度参数
keywordVarArgs(123,'abc',456)
```

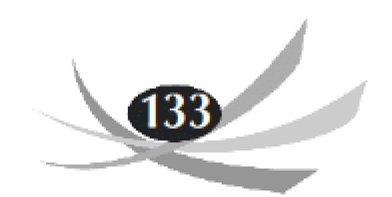
```
#指定位置参数、默认参数和两个非关键字的可变长度参数,同时指定一个关键字可变长度参数
keywordVarArgs(123,'abc',456,'def',kw1=1)
#指定位置参数、默认参数和三个非关键字的可变长度参数,同时指定两个关键字可变长度参数
keywordVarArgs(123,'abc',456,'def','xyz',kw1=1,kw2='a')
#指定位置参数、默认参数和三个非关键字的可变长度参数,同时指定三个关键字可变长度参数
keywordVarArgs(123,'abc',456,'def','xyz',kw1=1,kw2='a',kw3=[1,'a'])
```

程序的运行结果如下：

```
位置参数 posarg: 123
默认参数 defarg: abc
非关键字可变长度参数 vararg1: 456
--------------------------------------------------
位置参数 posarg: 123
默认参数 defarg: abc
非关键字可变长度参数 vararg1: 456
非关键字可变长度参数 vararg2: def
关键字可变长度参数 kw1: 1
--------------------------------------------------
位置参数 posarg: 123
默认参数 defarg: abc
非关键字可变长度参数 vararg1: 456
非关键字可变长度参数 vararg2: def
非关键字可变长度参数 vararg3: xyz
关键字可变长度参数 kw1: 1
关键字可变长度参数 kw2: a
--------------------------------------------------
位置参数 posarg: 123
默认参数 defarg: abc
非关键字可变长度参数 vararg1: 456
非关键字可变长度参数 vararg2: def
非关键字可变长度参数 vararg3: xyz
关键字可变长度参数 kw1: 1
关键字可变长度参数 kw3: [1, 'a']
关键字可变长度参数 kw2: a
--------------------------------------------------
```

该程序的第一个函数调用 keywordVarArgs(123,'abc',456)指定了位置参数、默认参数和一个非关键字的可变长度参数,但不指定关键字可变长度参数,所以可变长度参数 kwvarargs 是一个空的字典,没有输出。后面的三个函数调用分别指定了一、二和三个关键字可变长度参数。相应的,字典 kwvarargs 也分别存有一、二和三个参数值,遍历这个字典,输出其中的参数值。

需要说明的是：函数调用 keywordVarArgs(123,'abc',456,'def','xyz',kw1=1,kw2='a')还可以写成 keywordVarArgs(123,'abc', *(456,'def','xyz'), **{kw1: 1,kw2: 'a'})



或者是 keywordVarArgs(123,'abc',＊aTuple,＊＊aDict)(其中,aTuple 和 aDict 在函数调用前已分别赋值为(456,'def','xyz')和{kw1：1,kw2：'a'}),这三种方式的调用,其结果都是一样的。

6.3.6　函数返回值

通常,希望通过函数调用使主调函数能得到一个确定的值,这就是函数的返回值。例如,例 6-7 中,distance(5,10)的返回值为 125.0,并把它通过赋值语句赋给 result 变量。

函数的返回值是通过函数中的 return 语句获得的,return 语句可以返回一个值或者一个元组,当返回多个值时是把这些值先存到一个元组中,再把这个元组返回。实际上,还是返回一个对象,只是由于元组在语法上不需要一定带上圆括号,所以让人误以为可以返回多个对象。

一个函数中可以有一个以上的 return 语句,但最终只有一个 return 语句起作用。实际上,没有 return 语句,函数也有返回值,只是这个返回值是 None。

下面通过例 6-11 来理解函数的返回值。

```
#coding:utf-8
#例 6-11　关键字可变长度参数
def noReturnValue():
    print '没有指定返回值'

def returnOneValue(t):
    if(t):
        return True
    else:
        return False

def returnMoreValue():
    return 123,'abc',['xyz']

print noReturnValue()
print returnOneValue(1)
print returnMoreValue()
```

程序的运行结果如下：

```
程序的运行结果如下:
没有指定返回值
None
True
(123, 'abc', ['xyz'])
```

程序中的 noReturnValue 函数没有 return 语句,函数返回值为 None,所以输出 noReturnValue()就输出了 None。returnOneValue 函数根据参数的值的真假返回 true

或者 false,所以 returnOneValue(1)将返回 true。returnMoreValue 函数通过 return 123,'abc',['xyz']返回了三个值,实际上返回了一个包含三个值的元组。

6.4 函数属性和内嵌函数

6.4.1 函数属性

在 Python 中,每一个模块、类和函数都有一个属于自己的命名空间(命名空间是标识符到对象的映射,命名空间将在第 8 章中详细介绍)。函数属性是属于函数这个对象对应的命名空间,函数属性分固有属性(如__doc__和__name__等)和自定义属性,自定义属性是程序员根据需要向函数的命名空间添加的。引用函数属性的方式是:函数名.属性名,若 max 为已定义的函数,则可以通过 max.__doc__引用函数的__doc__属性。向函数添加自定义属性可以通过赋值语句实现。如语句 max.version=1.0 是向 max 函数添加自定义属性 version,且其值被赋值为 1.0。

```
#coding:utf-8
#例 6-12  函数属性
def func_attribute():
    'display the attribute of the function'
    pass

#输出初始化函数的固定属性
print '初始化函数的固定属性__doc__:',func_attribute.__doc__
print '初始化函数的固定属性__name__:',func_attribute.__name__
#对函数固定属性进行重新赋值
func_attribute.__name__='函数名'
func_attribute.__doc__='演示函数属性'
#输出重新赋值后的函数固定属性
print '重新赋值后函数的固定属性__doc__:',func_attribute.__doc__
print '重新赋值后函数的固定属性__name__:',func_attribute.__name__

#向函数添加自定义属性 version
func_attribute.version=1.0
print '函数的自定义属性 version:',func_attribute.version
```

程序的运行结果如下:

```
初始化函数的固定属性__doc__: display the attribute of the function
初始化函数的固定属性__name__: func_attribute
重新赋值后函数的固定属性__doc__: 演示函数属性
重新赋值后函数的固定属性__name__: 函数名
函数的自定义属性 version: 1.0
```

从程序的运行结果可以看出函数的固定属性的值可以被修改。

6.4.2　内嵌函数

在 Python 语言中,允许在一个函数内部再定义另外一个函数,这个在函数内部定义的函数就称为内嵌函数。但对于 C 语言,则不允许在函数内部再定义另外一个函数。定义内嵌函数的语法如下:

```
def func():
    func_statement
    def func_inner():
        func_inner_statement
```

其中,func_inner 就是内嵌函数,它是内嵌到 func 函数里面的。func 函数可以称为 func_inner 函数的外部函数。

注意,内嵌函数是在它的外部函数的命名空间下的,只有在它的外部函数的函数体内才能够调用该内嵌函数。

```
#coding:utf-8
#例 6-13　内嵌函数
def func_outer():
    '演示内部函数的使用'
    print 'func_outer 函数被调用'
    def func_inner():
        print 'func_inner 函数被调用'
    #在外部函数内调用 func_inner 内嵌函数
    func_inner()

func_outer()
#在外部函数外调用 func_inner 内嵌函数
func_inner()
```

程序的运行结果如下:

```
func_outer 函数被调用
func_inner 函数被调用

Traceback (most recent call last):
  File "C:/Python27/6.11.py", line 13, in<module>
    func_inner()
NameError: name 'func_inner' is not defined
```

该程序在 func_outer 外部函数的函数体内调用它的 func_inner 内嵌函数时能够正常调用,但在 func_outer 外部函数外调用,则抛 NameError 异常,因为内嵌函数只在它的外部函数的函数体内有效。

6.5 函数的嵌套调用

在 Python 语言中，函数不仅可以嵌套定义，还可以嵌套调用，即在调用一个函数的过程中又调用了另外一个函数。

图 6-6 表示函数的两层嵌套调用，其执行过程如下：

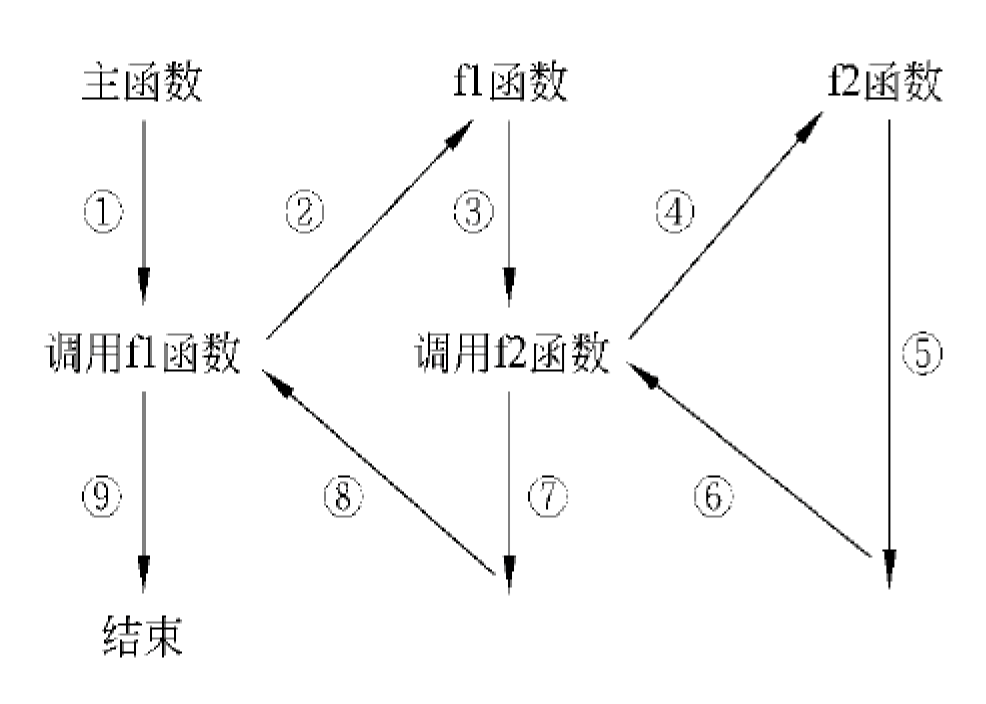


图 6-6 函数的两层嵌套调用

(1) 执行主函数的开头部分。

(2) 遇到函数调用，调用 f1 函数，流程转去 f1 函数。

(3) 执行 f1 函数的开头部分。

(4) 遇到函数调用，调用 f2 函数，流程转去 f2 函数。

(5) 执行 f2 函数，如果再无其他嵌套的函数，则完成 f2 函数的全部操作。

(6) 返回到 f1 函数中调用 f2 函数的位置。

(7) 继续执行 f1 函数中尚未执行的部分，直到 f1 函数结束。

(8) 返回主函数中调用 f1 函数的位置。

(9) 继续执行主函数的剩余部分直到结束。

下面通过例 6-14 来理解函数的嵌套调用，这个例子是用弦截法求方程 $f(x)=x^3-3x^2+8x-14=0$ 的根。弦截法的具体求法如下：

(1) 取两个不同的点 x_1、x_2，如果 $f(x_1)$ 和 $f(x_2)$ 符号相反，则在 (x_1,x_2) 区间内必有一个根。如果 $f(x_1)$ 和 $f(x_2)$ 符号相同，则应该改变 x_1、x_2，直到 $f(x_1)$、$f(x_2)$ 符号相反为止。同时还应该注意 x_1、x_2 的值不能相差太大，以保证在 (x_1,x_2) 区间内只有一个根。

(2) 连接 $((x_1,f(x_1))$ 和 $(x_2,f(x_2))$ 两点，此弦交 x 轴于 x，见图 6-7。

x 点的坐标可以用下式求出：

$$x=\frac{x_1 f(x_2)-x_2 f(x_1)}{f(x_2)-f(x_1)}$$

从而可以求得 f(x)。

(3) 若 f(x) 和 $f(x_1)$ 符号相同，则根必在 (x,x_2) 区间内，此时将 x 作为新的 x_1。如果 f(x) 和 $f(x_2)$ 符号相同，

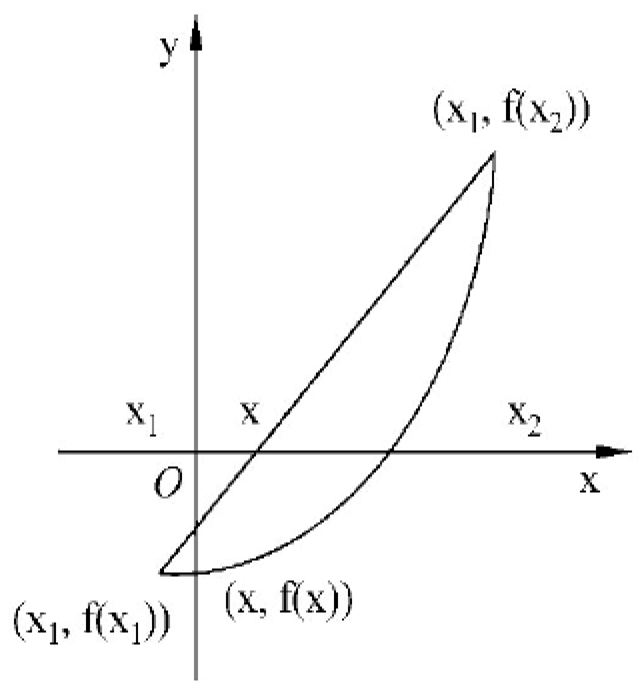


图 6-7 弦截法示意图

则根必在(x_1,x)区间内,此时将x作为新的x_2。

(4) 重复步骤(2)和(3),直到$|f(x)|<\varepsilon$为止,ε为一个很小的数,例如10^{-6}。此时可以认为$f(x)\approx 0$。

根据上面的思路,这个程序可以定义以下几个函数:

(1) f(x)函数。表示求x的函数值:$x^3-3x^2+8x-14$。

(2) xpoint(x_1,x_2)函数。表示求((x_1,f(x_1))和(x_2,f(x_2))的连线与x轴的交点x。

(3) root(x_1,x_2)函数。表示求(x_1,x_2)区间内的实根。显然,执行root函数的过程中要调用xpoint函数,而执行xpoint函数的过程中又调用f函数。

程序代码如下:

```
#coding:utf-8
#例6-14　函数的嵌套调用
import math
def f(x):
    '求x的函数值:x³-3x²+8x-14'
    y=((x-3)*x+8)*x-14
    return y

def xpoint(x1,x2):
    '求((x1,f(x1))和(x2,f(x2))的连线与x轴的交点x'
    y=float((x1*f(x2)-x2*f(x1)))/(f(x2)-f(x1))
    return y

def root(x1,x2):
    '求(x1,x2)区间内的实根
    y1=f(x1)
    x=xpoint(x1,x2)
    y=f(x)
    #若x的函数值y的绝对值≥10e-6,则继续循环
    while math.fabs(y)>=10e-6:
        #f(x)和f(x1)符号相同
        if y*y1>0:
            x1=x
            y1=y
        else:
            x2=x
        x=xpoint(x1,x2)
        y=f(x)
    return x

x1,x2=input('输入两个点x1,x2:')
f1=f(x1)
f2=f(x2)
```

```
#若输入的两个点对应的函数值符号相同,则继续循环输入
while f1*f2>=0:
    x1,x2=input('输入两个点 x1,x2:')
    f1=f(x1)
    f2=f(x2)
x=root(x1,x2)
print '所求的根为%f' %x
```

若输入的值分别为 1,4,则程序的运行结果如下:

```
输入两个点 x1,x2:1,4
所求的根为 2.228859
```

该程序首先分别定义了 f、xpoint、root 这三个函数,然后在主函数中输入 x1 和 x2,求 f(x1)和 f(x2)的值并判断 f(x1)和 f(x2)是否异号。如果不是异号,则重新输入 x1 和 x2,直到 f(x1)和 f(x2)异号为止。然后调用 root(x1,x2)求根 x。调用 root 函数过程中,需要调用 xpoint 函数求((x1,f(x1))和(x2,f(x2))的连线与 x 轴的交点 x。在调用 xpoint 函数的过程中需要调用 f 函数来求 x1 和 x2 相应的函数值 f(x1)和 f(x2)。

该程序的函数嵌套调用如图 6-8 所示。

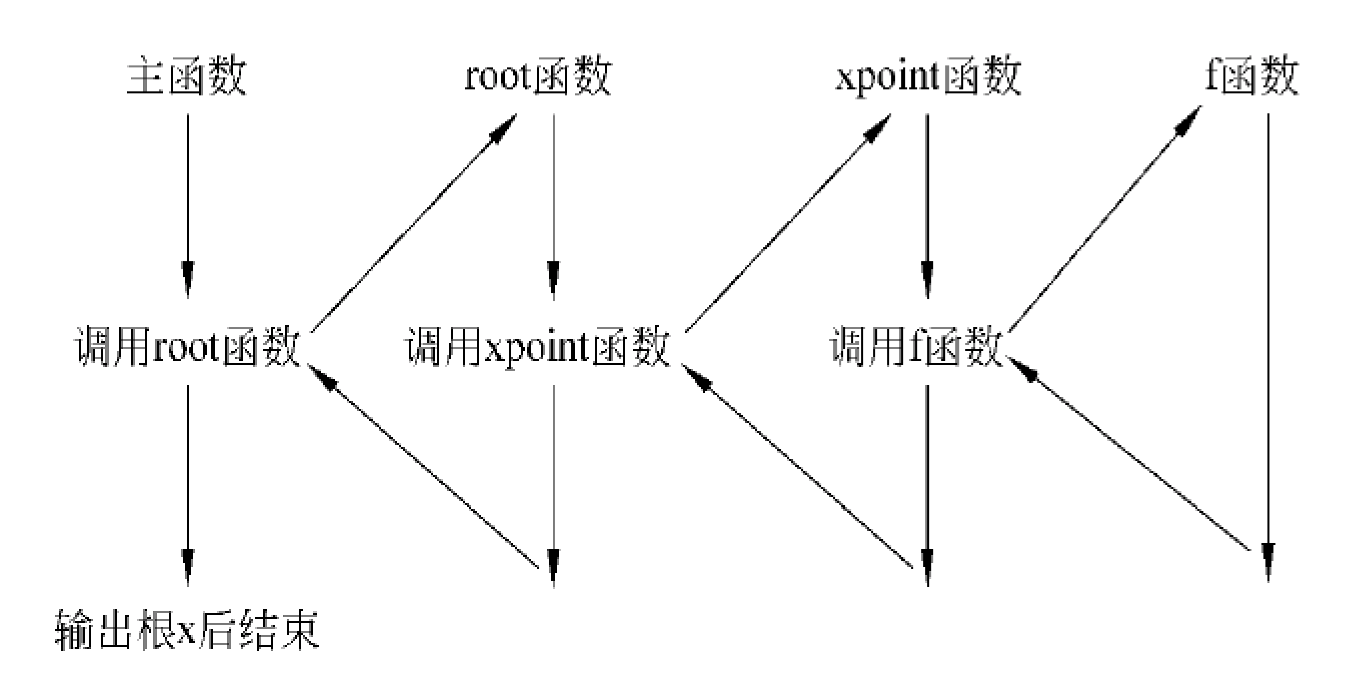


图 6-8　例 6-14 中函数的嵌套调用

6.6　函数的递归调用

在调用一个函数的过程中又出现直接或间接调用该函数本身,称为函数的递归调用。图 6-9 和图 6-10 分别展示了函数的直接递归调用和间接递归调用。

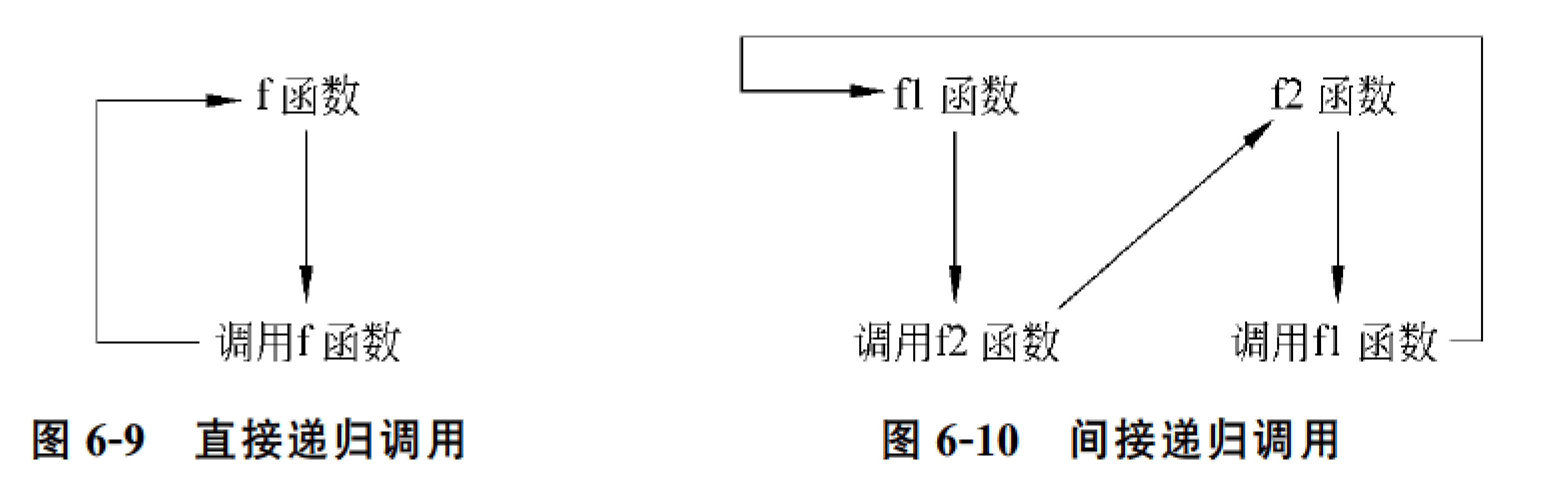


图 6-9　直接递归调用　　**图 6-10　间接递归调用**

从图 6-9 和图 6-10 可以看出,这两种递归调用都是无终止的自身调用。显然,程序

中不能出现这种无终止的递归调用,而只应出现有限次的、有终止的递归调用,这可以通过 if 条件语句来控制,只有在某个条件成立时才继续执行递归调用,否则就不再继续调用。

下面先通过一个通俗的例子来说明递归的概念。

有 5 个人参加运动会,他们分别被编成 1~5 号,现在问 5 号在这次运动会中总共获得了多少分?他说比 4 号获得的分还多 3 分。当问 4 号获得了多少分时,他说比 3 号获得的分还多 3 分。当问 3 号获得了多少分时,他说比 2 号获得的分还多 3 分。当问 2 号获得了多少分时,他说比 1 号获得的分还多 3 分。当问 1 号获得了多少分时,他说获得了 12 分。请问 5 号在这次运动会中获得了多少分。

显然,这是一个递归问题。要求 5 号在运动会中获得了多少分,就必须要先知道 4 号获得了多少分,而 4 号也不知道获得了多少分,要求 4 号获得的分数就必须先知道 3 号获得了多少分,而 3 号获得的分数又取决于 2 号获得的分数,2 号获得的分数又取决于 1 号获得的分数。而且每号对应的人获得的分数都比他小一号的人获得的分数多 3 分。即:

score(5)=score(4)+3

score(4)=score(3)+3

score(3)=score(2)+3

score(2)=score(1)+3

score(1)=12

可以用数学公式表述如下:

$$\mathrm{score}(n)=\begin{cases}12 & n=1\\ \mathrm{score}(n-1)+3 & n>1\end{cases}$$

可以看到,当 n>1 时,求第 n 号对应的人获得的分数的公式是相同的。因此可以用一个函数表示上述关系。图 6-11 表示求第 5 号对应的人获得的分数。

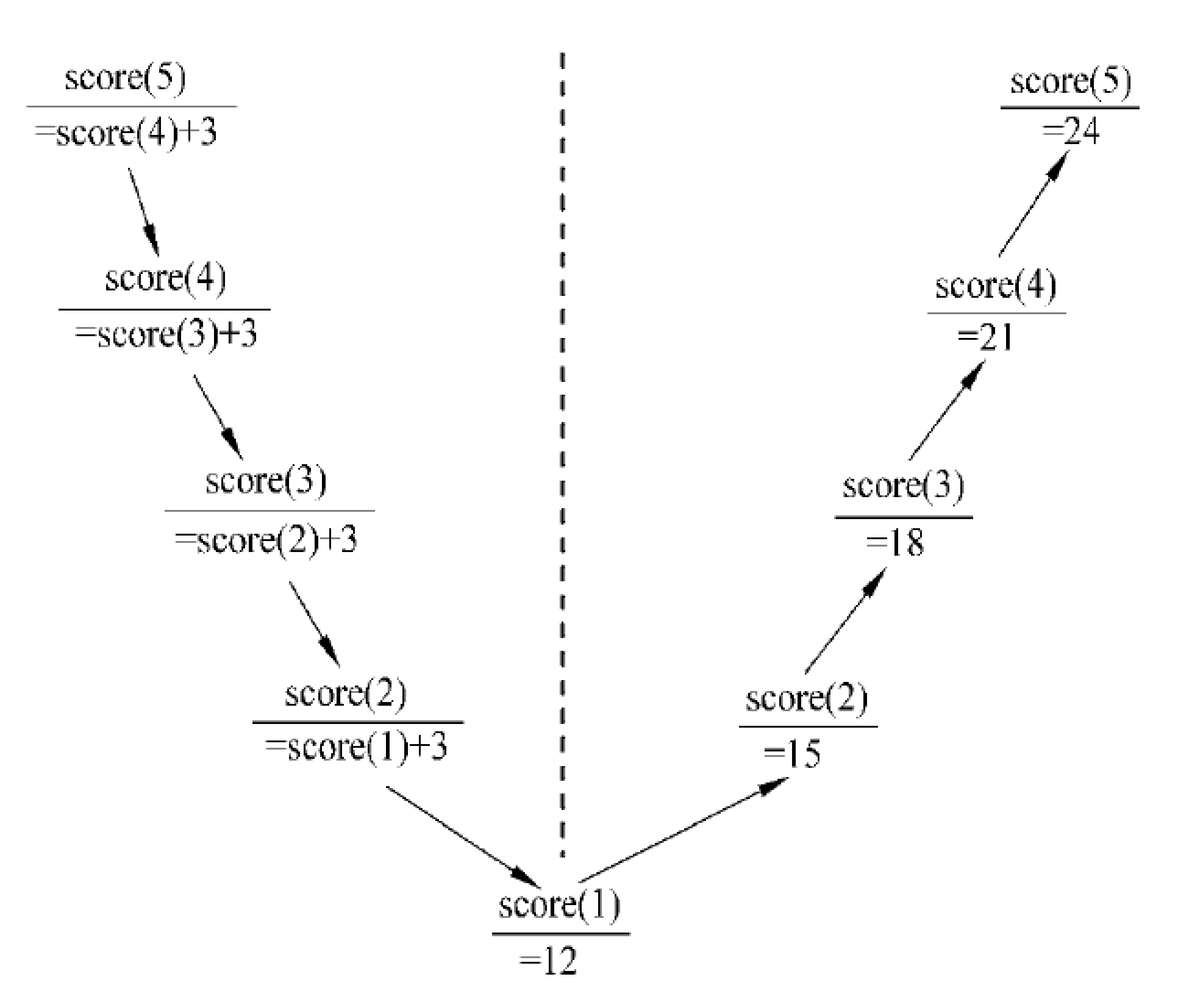


图 6-11 求第 5 号对应的人获得的分数的过程

从图 6-11 可知,求解可分成两个阶段:第一阶段是“回推”,即将第 n 号对应的人获

得的分数表示为第 n－1 号人获得的分数的函数，而第(n－1)号对应的人获得的分数仍然不知道，还要“回推”到第 n－2 号人获得的分数……直到第 1 号对应的人获得的分数。此时 score(1)已知，为 12 分，所以不必再“回推”。然后开始第二阶段，采用递推方法，从第 1 号对应的人获得的分数推算出第 2 号人获得的分数(15 分)，从第 2 号人获得的分数推算出第 3 号人获得的分数(18 分)……一直推算到第 5 号人获得的分数(24 分)为止。即一个递归的问题可以分为“回推”和“递推”两个阶段。要经历若干步才能求出最后的值。显然，递归过程必须要一个结束递归过程的条件，而不是无限制地进行下去。例如，该例中的 score(1)＝12，就是使递归结束的条件。

这个例子的程序如下：

```
#coding:utf-8
#例 6-15  函数的递归调用
def score(n):
    if n==1:
        r=12
    else:
        r=score(n-1)+3
    return r
print '第 5 号对应的人获得的分数:',score(5)
```

程序运行结果如下：

```
第 5 号对应的人获得的分数:24
```

函数的调用过程如图 6-12 所示。

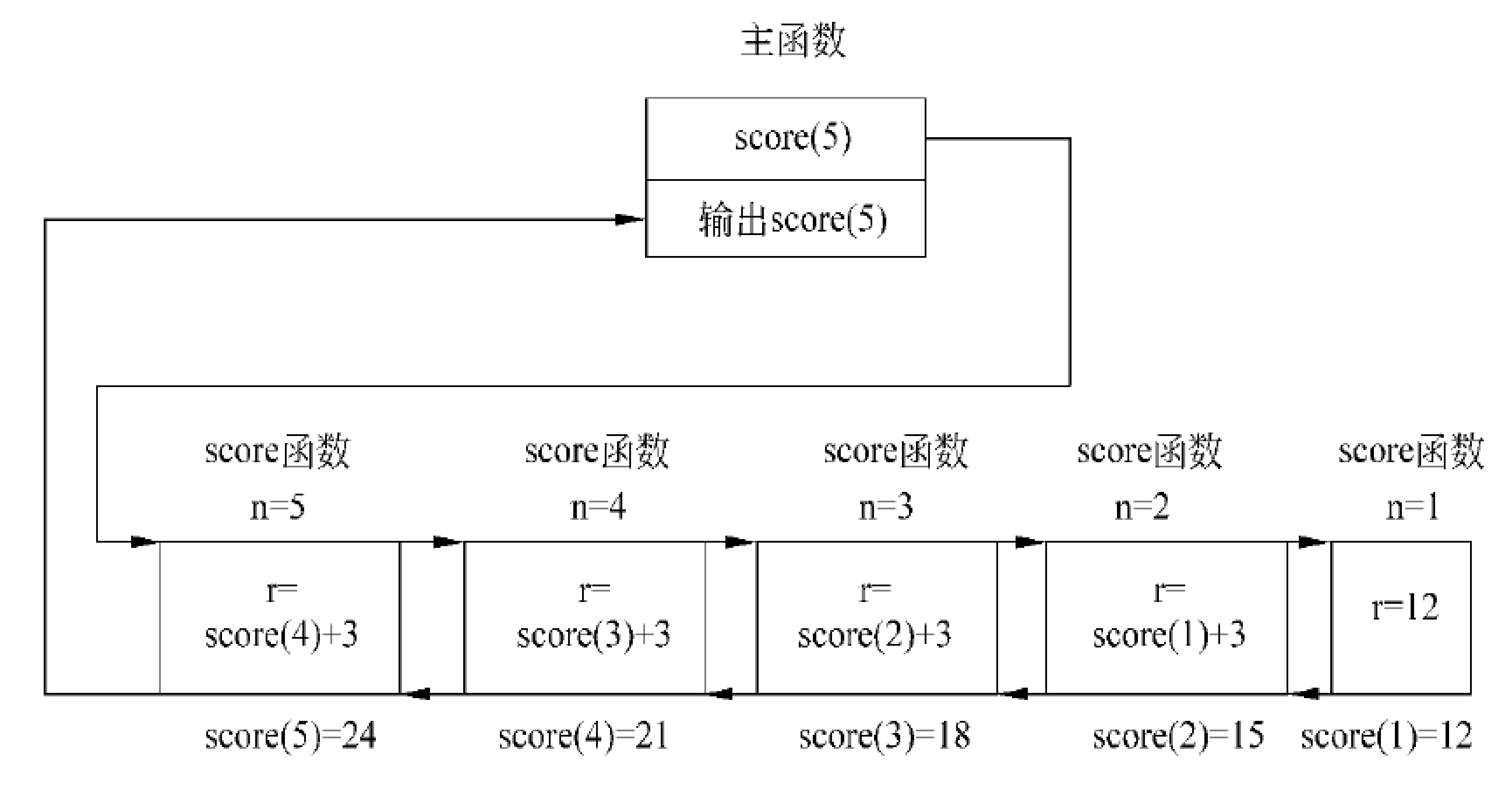


图 6-12　例 6-15 中函数的调用过程

从图 6-12 可以看到：score 函数共被调用了 5 次，即 score(5)、score(4)、score(3)、score(2)、score(1)。其中 score(5)是主函数调用的，其余 4 次是在 score 函数中被调用的，即递归调用了 4 次。应当注意，在某一次调用 score 函数时并不是立即得到 score(n)的值，而是一次又一次地进行递归调用，直到调用 score(1)时才得到确定的值(12)，然后

再递推出 score(2)、score(3)、score(4)、score(5)。

下面举一个很经典的递归例子。

Hanoi(汉诺)塔问题。这是一个古典的数学问题,是一个用递归方法解题的典型例子。这个问题是这样的:古代有一个梵塔,塔内有三个座 A、B、C,开始时 A 座上有 64 个盘子,盘子大小不等,大的在下,小的在上(见图 6-13)。有一个老和尚想把这 64 个盘子从 A 座移到 C 座,但每次只允许移动一个盘,且在移动过程中在三个座上都始终保持大盘在下,小盘在上。在移动过程中可以利用 B 座,要求编程序打印出移动的步骤。

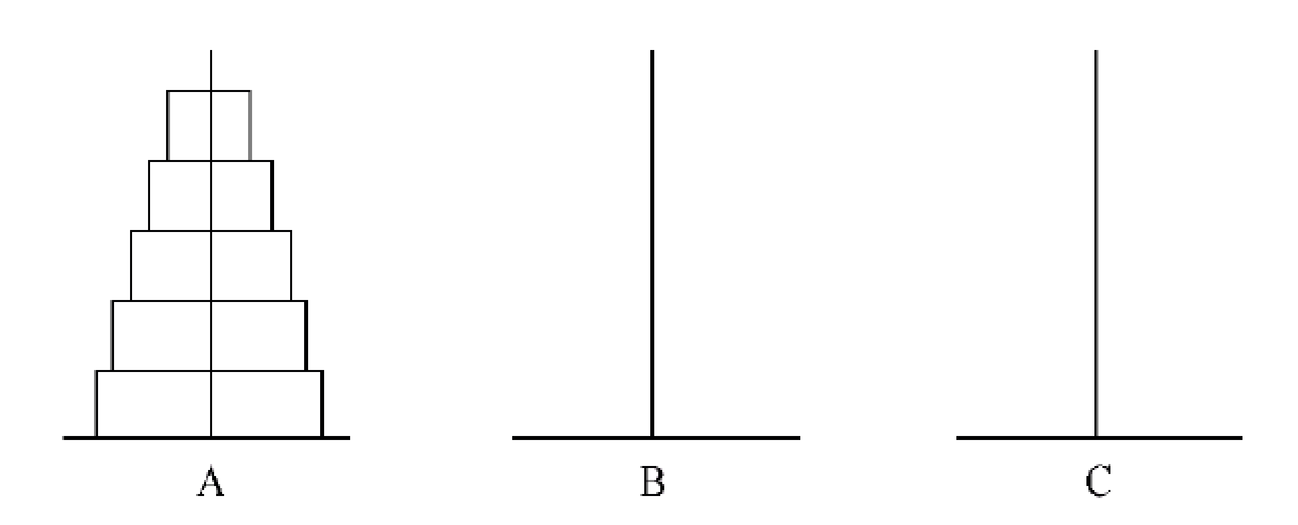


图 6-13 汉诺塔问题示意图

老和尚就想:假如有另外一个和尚能有办法将 63 个盘子从一个座移到另一个座上。那么,问题就解决了。此时老和尚只需这样做:

(1) 吩咐第 2 个和尚将 63 个盘子从 A 座移到 B 座。

(2) 自己将最底下的、最大的盘子从 A 座移到 C 座。

(3) 再吩咐第 2 个和尚将 63 个盘子从 B 座移动到 C 座。

至此,全部任务都已完成。实际上,上面有一个问题还没有解决,即第 2 个和尚怎样才能将 63 个盘子从 A 座移动到 B 座?

为了解决将 63 个盘子从 A 座移到 B 座,第 2 个和尚又想:如果有第 3 个和尚能将 62 个盘子从一个座上移到另一座上,我将能将 63 个盘子从 A 座移到 B 座,他是这样做的:

(1) 吩咐第 3 个和尚将 62 个盘子从 A 座移到 C 座。

(2) 自己将最底下的、最大的盘子从 A 座移到 B 座。

(3) 再吩咐第 3 个和尚将 62 个盘子从 C 座移动到 B 座。

这样“层层下放”,直到最后找到第 64 个和尚,让他完成将 1 个盘子从一个座上移到另一座上,至此,全部任务才真正完成,这就是递归方法。

可以看出,递归的结束条件是最后一个和尚只需移动一个盘子;否则递归还要继续进行下去。

应当说明的是,只有第 64 个和尚的任务完成后,第 63 个和尚的任务才能完成。只有第 2 个到第 64 个和尚的任务都完成后,第 1 个和尚的任务才能完成。这是一个典型的递归问题。

为便于理解,先分析将 A 座上 3 个盘子移到 C 座上的过程:

(1) 将 A 座上 2 个盘子借助 C 座移到 B 座上。

(2) 将 A 座上 1 个盘子移到 C 座上。

(3) 将 B 座上 2 个盘子借助 A 座移到 C 座上。

其中第(2)步可以直接实现。第(1)步又可用递归方法分解为：

1.1 将 A 座上 1 个盘子移到 C 座上。

1.2 将 A 座上 1 个盘子移到 B 座上。

1.3 将 C 座上 1 个盘子移到 B 座上。

第(3)步同样也可用递归方法分解为：

3.1 将 B 座上 1 个盘子移到 A 座上。

3.2 将 B 座上 1 个盘子移到 C 座上。

3.3 将 A 座上 1 个盘子移到 C 座上。

综上所述：可得到移动 3 个盘子的步骤为：A→C，A-B，C→B，A→C，B→A，B→C，A→C。共需要 7 步。由此可推出：移动 n 个盘子需要 2^n-1 步。如移动 5 个盘子需要 31 步，移动 64 个盘子需要 $2^{64}-1$ 步。

由上面的分析可知：将 n 个盘子从 A 座移到 C 座可以分解为以下 3 个步骤：

(1) 将 A 座上 n−1 个盘子借助 C 座移到 B 座上。

(2) 将 A 座上 1 个盘子移到 C 座上。

(3) 将 B 座上 n−1 个盘子借助 A 座移到 C 座上。

上面第(1)步和第(3)步都是把 n−1 个盘子从一个座上移到另一座上，只是座的名字不同而已。因此，可以将第(1)步和第(3)步表示为：

将 one 座上 n−1 个盘子借助 three 座移到 two 座。只是在第(1)步和第(3)步中，one，two，three 和 A，B，C 的对应关系不同。对第(1)步，对应关系是 one 对应 A，two 对应 B，three 对应 C。对第(3)步，就是 one 对应 B，two 对应 C，three 对应 A。

由此，可以把上面 3 个步骤分成两类操作：

(1) 将 n−1 个盘子从一个座上移到另一座上(n>1)。这是一个递归的过程，即和尚将任务层层下放，直到第 64 个和尚为止。

(2) 将 1 个盘子从一个座上移到另一座上。

根据上面的分析可以编写程序。上面的两类操作分别用两个函数实现，用 hanoi 函数实现第(1)类操作，用 move 函数实现第(2)类操作。函数调用 hanoi(n，one，two，three)表示将 n 个盘子从 one 座借助 two 座移到 three 座的过程。函数调用 move(x，y)表示将 1 个盘子从 x 座移到 y 座的过程。其中，x 和 y 是代表 A，B，C 座之一，根据每次不同情况分别取 A、B、C 代入。

```
#coding:utf-8
#例 6-16  函数的递归调用(汉诺塔问题)
def hanoi(n,one,two,three):
    if n==1:
        move(one,three)
    else:
        hanoi(n-1,one,three,two)
        move(one,three)
        hanoi(n-1,two,one,three)
```

```
def move(x,y):
    print '%c->%c' % (x,y)
n=input('请输入盘子的数目:')
print '移动'+str(n)+'个盘子的步骤如下:'
hanoi(n,'A','B','C')
```

如果输入是3,则运行结果如下:

```
请输入盘子的数目:3
移动 3个盘子的步骤如下:
A->C
A->B
C->B
A->C
B->A
B->C
A->C
```

对于n个盘子,本程序输出了移动盘子的方案,即依次输出从哪一个座上移动盘子到哪个座上。注意,n要小,如果n大了,执行完该程序需要耗费很多的时间。

可以看到,将3个盘子从A座移到C座需要移动7次,如果将64个盘子从A座移到C座需要移 $2^{64}-1$ 次,假设和尚每次移动一个盘子用1秒钟,则移动 $2^{64}-1$ 次需要 $2^{64}-1$ 秒,大约相当于 6×10^{11} 年,即大约600亿年。

6.7 变量的作用域

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里定义(赋值)的。

变量的作用域决定了在哪一部分程序可以访问哪个特定的变量名称。两种最基本的变量作用域如下:

- 局部变量;
- 全局变量。

下面分别介绍这两种变量作用域。

6.7.1 局部变量

定义在函数内部的变量称为局部变量,局部变量只有在此函数内有效,也就是说,只有在此函数内才能使用它们。在此函数外的所有地方都不能使用它们。例如:

```
#coding:utf-8
#例 6-17  局部变量
def f1(a):
    b,c=2,3                          #a,b,c只有在此处到 f1函数结束前有效
```

```
    print "f1函数中的a,b,c:",a,b,c
def f2(x,y):
    z=3                                 #x,y,z只有在此处到f2函数结束前有效
    b=4                                 #b只有在此处到f2函数结束前有效
    print "f2函数中的x,y,z,b:",x,y,z,b
    print c                             #试图使用f1函数中的变量c
f1(1)
f2(1,2)
```

程序运行结果如下：

```
f1函数中的a,b,c: 1 2 3
f2函数中的x,y,z,b: 1 2 3 4

Traceback (most recent call last):
  File "C:/Python27/6.17.py", line 13, in<module>
    f2(1,2)
  File "C:/Users/wangs/Desktop/6.17.py", line 10, in f2
    print c                             #试图使用f1函数中的变量c
NameError: global name 'c' is not defined
```

从程序的运行结果可以看出，在f1函数内定义的局部变量a,b,c可以在该函数内正常使用，在f2函数内定义的局部变量x,y,z,b可以在该函数内正常使用，但在f2函数内使用f1函数定义的局部变量c，则抛NameError异常，提示b未定义。

说明：

（1）不同函数中可以使用相同名字的变量，它们代表不同的对象，相互之间不会造成影响。例如，例6-17中f1函数中的局部变量b，f2函数中的局部变量b，这两个变量有不同的存储空间，代表不同的对象。在f2中输出的是该函数的变量b，和f1函数中的变量b无关。

（2）形式参数也是局部变量，例如，例6-17中f1函数中的形参a，它也只有在f1函数内有效。其他函数可以调用f1函数，但不能引用f1函数中的形参a。

（3）局部变量的作用域从定义处到它所属的函数结束，即在定义处到它所属的函数结束前有效。例如，例6-17中，在f2函数内，赋值语句“b=4”前不能引用局部变量b，否则将抛UnboundLocalError异常，提示局部变量t在声明t前被引用。

（4）函数被调用时，该函数中的局部变量才被分配到存储单元。函数一旦执行完，这些局部变量的存储单元就被释放。也就是说，局部变量的生命周期（存在时间）从函数调用到函数执行完毕。

6.7.2 全局变量

前一节已介绍，定义在函数内部的变量称为局部变量，那什么是全局变量呢？我们把定义在函数外部的变量称为全局变量。全局变量作用域和局部变量的作用域相似，都是从定义处到它所属的函数结束，不同的是，其他函数（包括在定义全局变量前所定义的函

数)都可以引用全局变量。

在 Python 中,对于标识符(变量)的搜索,是先从局部作用域内开始搜索,如果在局部作用域内没有找到给定的标识符,则到全局作用域内查找,如果还是没找到,就会抛 NameError 异常。可见,当某函数内定义的局部变量和全局变量同名时,在这个函数内,与局部变量同名的全局变量将不起作用,或者说是被“屏蔽”了。当然,可以通过使用 global 关键字来明确声明使用全局变量。

全局变量的生命周期从程序启动(系统调用主函数)到该程序执行完毕。

下面通过例 6-18 和例 6-19 来理解全局变量:

```
#coding:utf-8
#例 6-18　全局变量
total=0
def sum(a,b):
    "返回两个参数的和."
    print "sum 函数中引用全局变量 total:", total
    print "sum 函数中引用全局变量 maxv:", maxv
    t=a+b
    return t

maxv=0
def max(a,b):
    "返回两个参数中的较大者."
    print "max 函数中引用全局变量 total:", total
    print "max 函数中引用全局变量 maxv:", maxv
    #下面的 maxv 是局部变量
    if a>b:
        maxv=a
    else:
        maxv=b
    print "max 函数中的局部变量 maxv:", maxv
    return maxv

sum(10,20)
max(10,20)
print "函数外引用全局变量 total:", total
print "函数外引用全局变量 maxv:", maxv
```

程序运行结果如下:

```
sum 函数中引用全局变量 total: 0
sum 函数中引用全局变量 maxv: 0
max 函数中引用全局变量 total: 0
max 函数中引用全局变量 maxv:
```

```
Traceback (most recent call last):
  File "C:/Python27/6.18.py", line 26, in<module>
    maxv=max(10,20)
  File "C:/Users/wangs/Desktop/6.18.py", line 15, in max
    print "max函数中引用全局变量 maxv:", maxv
UnboundLocalError: local variable 'maxv' referenced before assignment
```

从程序的运行结果可以看出，在 sum 函数内可以引用全局变量 total 和 maxv，它们的初始值都是 0，调用完 sum 函数后，接着调用 max 函数，在 max 函数中，首先输出全局变量 total，其值仍为 0，然后试图输出全局变量 maxv。可以发现此时抛 UnboundLocalError 异常，提示局部变量 maxv 在声明前被引用。这是由于在 max 函数中声明的局部变量和全局变量同名，使得全局变量在 max 函数中不起作用，或者说被"屏蔽"了。这样，在 max 函数中输出的变量 maxv 其实是局部变量，又由于引用局部变量 maxv 在未声明它之前，所以就抛 UnboundLocalError 异常。

现在把 max 函数中的第二条 print 语句注释掉，程序的运行结果如下：

```
sum函数中引用全局变量 total: 0
sum函数中引用全局变量 maxv: 0
max函数中引用全局变量 total: 0
max函数中的局部变量 maxv: 20
函数外引用全局变量 total: 0
函数外引用全局变量 maxv: 0
```

当然，可以在 max 函数内使用 global 关键字明确声明使用全局变量。把 max 函数修改成如下所示：

```
def max(a,b):
    "返回两个参数中的较大者."
    #使用 global 关键字声明 maxv 为全局变量
    global maxv
    print "max函数中引用全局变量 total:", total
    print "max函数中引用全局变量 maxv:", maxv
    if a>b:
        maxv=a
    else:
        maxv=b
    print "赋值后的全局变量 maxv:", maxv
    return maxv
```

程序的运行结果如下：

```
sum函数中引用全局变量 total: 0
sum函数中引用全局变量 maxv: 0
max函数中引用全局变量 total: 0
max函数中引用全局变量 maxv: 0
```

```
赋值后的全局变量 maxv: 20
函数外引用全局变量 total: 0
函数外引用全局变量 maxv: 20
```

可以看到，在 max 函数中，全局变量 maxv 被指向新创建的对象 20，在函数外引用的全局变量 maxv 已是新指向的对象 20。

注意：这里的全局变量是整型变量，即它指向的对象是不可变的类型，但对象的引用(指向)可以改变。

下面看看全局变量是可变类型(列表)的情况：

```
#coding:utf-8
#例 6-19　全局变量(可变类型)
total=[0]
def sum(a,b):
    "返回两个参数的和."
    print "sum 函数中引用全局变量 total:", total
    print "sum 函数中引用全局变量 maxv:", maxv
    #修改全局变量 total 的值
    total[0]=a+b

maxv=[0]
def max(a,b):
    "返回两个参数中的较大者."
    print "max 函数中引用全局变量 total:", total
    print "max 函数中引用全局变量 maxv:", maxv
    #修改全局变量 maxv 的值
    if a>b:
        maxv[0]=a
    else:
        maxv[0]=b

sum(10,20)
max(10,20)
print "函数外引用全局变量 total:", total
print "函数外引用全局变量 maxv:", maxv
```

程序运行结果如下：

```
sum 函数中引用全局变量 total: [0]
sum 函数中引用全局变量 maxv: [0]
max 函数中引用全局变量 total: [30]
max 函数中引用全局变量 maxv: [0]
函数外引用全局变量 total: [30]
函数外引用全局变量 maxv: [20]
```

该程序在 sum 函数中首先输出全局变量 total 和 maxv，然后修改 total 列表中的第

一个元素,使其等于 sum 函数中的两个形参的和,所以在 max 函数中输出全局变量 total,此时已是修改过的值“[30]”,然后修改全局变量 maxv 的第一个元素,使其等于 max 函数中两个形参的较大者,最后在函数外输出全局变量 total 和 maxv,它们都是已修改过的值,分别为“[30]”和“[20]”。

6.8 本章小结

本章主要讲解了以下几个知识点:

(1) Python 程序的结构。Python 程序是由包(package)、模块(module)、函数(function)组成的。其中,包是由一系列模块组成的集合,而模块是处理某一类问题的函数或(和)类的集合。

(2) 函数以及函数的分类。函数是组织好的,可重复使用的,用来实现单一,或相关联功能的代码段。函数能减少程序的代码量,节约了存储空间,同时也提高应用的模块性,代码的重复利用率和程序的可维护性。

(3) 函数的分类。从用户使用的角度来看,可以分为内建函数(由系统提供,用户可以直接使用)和用户自定义函数(由用户根据需要自己定义的用于解决用户特定问题的函数);从函数的形式看,可以分为无参函数和有参函数。这里的有参无参指的是函数定义中函数名后的括号内有无参数。

(4) 函数参数。在定义有参函数时函数名后面括号中的变量名就称为“形式参数”(简称“形参”),在调用有参函数时,函数名后面括号中的参数称为“实际参数”(简称“实参”)。形式参数又分为位置参数(以正确的位置顺序传入函数的参数)、关键字参数(以顺序或不按顺序传入,但带有参数列表中曾定义过的关键字)、默认参数(函数调用时不一定要指定的参数)和可变长度参数(每次函数调用时的参数数目允许不相同)。

(5) 参数传递。实参向形参的参数传递都是采用引用传递的方式。在函数调用时,实参传递引用给形参,使得实参和形参都指向相同的对象。当实参变量所指向的对象是可变对象时,改变形参所指向的对象实际上也改变了实参所指向的对象。当实参变量所指向的对象是不可变对象时,改变形参变量,如重新赋值,但由于形参变量所指向的对象不可变,所以形参变量将指向新的对象,而实参变量还是指向之前的对象。

(6) 函数属性和内嵌函数。函数属性是属于函数这个对象对应的命名空间,函数属性分固有属性和自定义属性,自定义属性是程序员根据需要向函数的命名空间添加的。在 Python 语言中,允许在一个函数内部再定义另外一个函数,这个在函数内部定义的函数就称为内嵌函数。内嵌函数只有在它的外部函数的函数体内才能够调用该内嵌函数。

(7) 函数的嵌套调用。函数的嵌套调用就是在调用一个函数的过程中又调用了另外一个函数。函数允许有多层嵌套调用。

(8) 函数的递归调用。函数的递归调用就是在调用一个函数的过程中又出现直接或间接调用该函数本身。递归调用必须存在一个条件,当满足该条件时,函数不再递归执行。

(9) 变量的作用域。变量的作用域决定了在哪一部分程序可以访问哪个特定的变量

名称。变量作用域分为局部变量和全局变量。局部变量是定义在函数内部的变量,局部变量只有在此函数内有效。它的作用域从定义处到它所属的函数结束。它的生命周期(存在时间)从函数调用到函数执行完毕。全局变量是定义在函数外部的变量,其他函数内和主函数内定义处后面的范围称为全局变量的作用域,在作用域范围内都可以引用全局变量。全局变量的生命周期从程序启动(系统调用主函数)到该程序执行完毕。

6.9 习 题

一、解答题

1. Python 程序的结构是怎样的?
2. 函数的分类是怎样的?
3. 函数参数的分类是怎样的? 形式参数又可以分为哪些?
4. 参数传递的方式是怎样的? 当实参变量所指向的对象分别是可变类型和不可变类型时,改变形参变量会有什么影响?
5. 什么是函数的嵌套调用? 什么是函数的递归调用?
6. 变量作用域分为哪些? 它们有什么特点?

二、看程序写结果

1.

```
def f1(posarg,defarg='Python',*varargs,**kwvarargs):
    print 'posarg:',posarg
    print 'defarg:',defarg
    i=1
    for eachvar in varargs:
        print 'vararg'+str(i)+':',eachvar
        i+=1
    i=1
    for key in kwvarargs:
        print 'kwvarargs'+key+':',kwvarargs[key]
        i+=1
    print

f1(123,'abc')
f1('abc',123)
f1(defarg='abc',posarg=123)
f1(123)
f1(123,'abc','def',[456,'xyz'])
f1(123,456,'def',kw1=1,kw2='a')
f1(123,'abc',*(456,'def'),**{'kw1':1,'kw2':'a'})
```

2.

```
def f(n):
```

```
    if n==1:
        r=1
    else:
        r=f(n-1)*n
    return r

print f(10)
```

3.

```
def inputdata():
    alist=[]
    global n
    n=input('请输入n行n列数据中n的大小:')
    i=0
    while i<n:
        atuple=input('请输入'+str(n)+'个整型数据,以逗号相隔:')
        alist.append(atuple)
        i+=1
    return alist

def maxloc(t):
    loc=0
    maxv=t[0]
    i=0
    for a in t:
        if a>maxv:
            loc=i
            maxv=a
        i+=1
    return loc

def ismin(m,t):
    flag=0
    minv=t[0]
    i=0
    for a in t:
        if a<minv:
            minv=a
        i+=1
    if m==minv:
        flag=1
    return flag

n=0
```

```
alist=inputdata()
j=0
flag=0
while j<n:
    maxi=maxloc(alist[j])
    vlist=[]
    i=0
    while i<n:
        vlist.append(alist[i][maxi])
        i+=1
    maxv=alist[j][maxi]
    if ismin(maxv,vlist)==1:
        print 'alist[%d][%d]=%d' % (maxi,j,maxv)
        flag=1
    j+=1
if flag==0:
    print 'There is no output!'
```

四、上机练习

1. 编写两个函数，分别求两个整数的最大公约数和最小公倍数，在主函数调用这两个函数，并输出结果。两个整数由键盘输入。

2. 编写两个函数 sum 和 fac，在 sum 函数中产生三个小于 10 的随机数，对这三个随机数分别调用 fac 函数求它们的阶乘，在 sum 函数中返回这三个随机数的阶乘的和，在主函数调用 sum 函数(提示：导入 random 模块中的 randint)。

3. 求方程 $ax^2+bx+c=0$ 的根，用三个函数分别求当 b^2-4ac 大于 0、等于 0 和小于 0 时的根并输出结果。从主函数输入 a、b、c 的值。

4. 编写一个函数，将一个十进制整数 n 转换为二进制并输出，n 由键盘输入。

5. 编写一个判断素数的函数，在主函数输入一个整数，调用该函数，输出是否是素数的信息。

6. 编写一个递归函数实现 Fibonacci 数列。它的递归公式如下所示：

$$F(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

7. 用递归的方法求 n 阶勒让德多项式的值，递归公式为如下所示：

$$P_n(x)=\begin{cases}1 & n=0\\ x & n=1\\ ((2n-1)*x-P_{n-1}(x)-(n-1)*P_{n-2}(x))/n & n\geqslant 1\end{cases}$$

第 7 章 面向对象编程

本章学习目标

- 理解面向对象程序设计的基本思路和主要特点
- 掌握类、对象以及它们之间的关系
- 掌握类、对象的属性和方法
- 了解类的内置属性和方法
- 掌握类的组合
- 掌握类的继承与派生
- 了解新式类的高级特性

本章将介绍新的内容——面向对象编程。这部分的内容和前面的内容有比较大的差异,前面的内容主要体现面向过程的程序设计,而这章介绍的内容是新的程序设计方法——面向对象的程序设计,这种设计方法与传统的面向过程的方法相比,具有更好的可重用性、可扩展性和可管理性。

面向对象程序设计的基本思路是:将数据和对数据的操作方法集中放在一个整体中,形成一个相互依存、不可分割的整体,这个整体即为对象。通过相同类型的对象,抽象出其共性而形成类。为了类能够与外界发生联系,在类中必须声明一些函数(方法),这些函数用于与外界进行通信。

7.1 概述

7.1.1 什么是面向对象的程序设计

面向对象程序设计的方法是从人们日常生活中处理问题的思路中启发而形成的一个新的设计方法,这样的设计方法更贴近我们的生活,让人更好地理解和掌握。

在自然界中,一个复杂的事物总是由许多部分组成的。例如,一台电脑是由 CPU、主板、内存条、硬盘、外壳等部件组成的;一个学校是由许多学院、行政部门、学生班级等组成的;一个城市是由许多地区、机构、人群等组成的。

当人们生成汽车时,并不是先设计和制造发动机,再设计和制造底盘,然后设计和制造机身和轮子,而是分别设计和制造发动机、底盘、车身和轮子,最后把它们组装在一起。在组装的过程中,各部分之间有一定的联系,以便协调工作。例如驾驶员踩油门,就能调

节油路，控制发动机的转速，驱动车轮转动。

这就是面向对象程序设计的基本思想。

为了更好地介绍这种设计方法，下面先介绍面向对象程序设计中几个很重要的术语。

1. 对象

从一般意义的角度上讲，对象是现实世界中实际存在的事物，包括一切有形的和无形的事物。对象可以是自然物体（如汽车、房屋、座子，水），也可以是社会生活中的一种逻辑结构（如班级、小组、部门），甚至还可以是一篇文章、一条新闻、一个短语。对象是世界中一个独立的单位，都有自己的特征，包括静态特征和动态特征。对象的静态特征可以用某些数据来描述，而动态特征表现为其所表现的行为或具有的功能。比如，一台录像机是一个对象，它的静态特征是生产厂家、牌子、重量、体积、颜色、价格等，这种静态特征称为属性；其动态特征是它的行为，根据外界给它的信息进行录像、放像、快进、倒退、暂停、停止等操作，这种特征称为行为。

对象是描述事物的一个实体，是构成程序的一个基本单位。对象由一组属性（数据）和一组行为（函数或称为方法）构成。属性用来描述对象的静态特征，行为用来描述对象的动态特征。

2. 类

抽象和分类是面向对象程序设计的两个原则。抽象是具体事物描述的一个概括，与具体是相对应的；而分类的依据是对对象共性的抽象。在对事物分类的过程中，忽略事物的非本质特征，只关注与当前对象有关的本质属性，从而提取事物的共性，把具有相同特性的事物划为一类，得出一个抽象的概念。比如，我们常用的名词“人”，就是一个抽象。因为世界上只有具体的人，如张三、李四、王五。把所有国籍为“中国”的人归纳为一类，称为“中国人”，这就是一种“抽象”。再把中国人、美国人、英国人等所有国家的人抽象为“人”。在实际生活中，我们只能看到一个个具体的人，而看不到抽象的人。

面向对象中的类是具有相同属性和行为的一组对象的集合，它能为全部对象提供抽象的描述，包括属性和行为。

类和对象的关系是抽象与具体的关系，它们的关系就像模具与用模具生产出来的产品的关系。一个属于某个类的对象称为该类的一个实例。

3. 封装

封装是面向对象程序设计方法的一个特点和重要原则。它是指将对象的属性和行为组合成一个独立的单元，并尽可能隐藏对象的内部细节。所以封装有两个特点，一个是将对象的全部属性和行为组合在一起，形成一个不可分割的独立单元（类）；二是需要对这个独立单元进行信息的隐藏，使得外界无法轻易获得单元中的信息，从而实现信息的保护，外界只有通过单元提供的某些特定接口（函数）与其发生联系。

比如，录像机里有电路板和机械控制部件，但是外面是看不到的，从外面看，它只是一个“黑箱子”，在它的表面有几个按键，这就是录像机与外界的接口，我们不必了解录像机里面的结构和工作原理，只需知道按某个键将能使录像机执行相应的操作即可。

4. 继承

继承是面向对象程序设计中能够提高程序的可重用性和开发效率的重要保障。一个

类的对象拥有另一个类的全部属性和行为,则可以将这个类声明为继承自另一个类。

如果在软件开发中已经建立了一个名为 A 的"类",又想另外建立一个名为 B 的"类",而后者与前者内容基本相同,只是在前者的基础上增加一些属性和行为,显然不必再从头设计一个新类,而只需在类 A 的基础上增加一些新内容即可。这就是面向对象程序设计中的继承机制。利用继承可以简化程序设计的步骤。举个例子:如果大家都已经充分认识了马的特征,现在要描述"白马"的特征,显然不必从头介绍什么是马,而只需说明"白马是白色的马"即可。这就简化了人们对事物的认识和叙述,简化了工作程序。

"白马"继承了"马"的基本特性,又增加了新的特征(颜色),"马"是父类,或称为基类,"白马"是从"马"派生出来的,称为子类或派生类。如果还想定义"白公马",只需说明"白公马是雄性的白马"。"白公马"又是"白马"的子类或派生类。

5. 多态性

如果有几个相似而不完全相同的对象,有时人们要求在向它们发出同一个消息时,它们的反应不相同,分别执行不同的操作。这种情况就是多态现象。例如甲、乙、丙三个都是高二年级,他们有基本相同的属性和行为,在同时听到上课铃声时,他们会分别走进三个不同的教室,而不会走向同一个教室。同样,如果有两支军队,当在战场上同时听到一种号声,由于事先约定不同,A 军队可能实施进攻,而 B 军队可能准备开饭。又如,在 Windows 环境下,用鼠标双击一个文件对象(这就是向对象传送一个消息),如果对象是一个可执行文件,则会执行此程序,如果对象是一个文本文件,则启动文本编辑器并打开该文件。

7.1.2 面向对象程序设计的特点

传统的面向过程的程序设计是围绕功能进行的,用一个函数来实现一个功能。所有的数据都是公用的,一个函数可以使用任何一组数据,而一组数据又能被多个函数所使用,如图 7-1 所示。程序设计者必须考虑每一个细节,什么时候对什么数据进行操作。

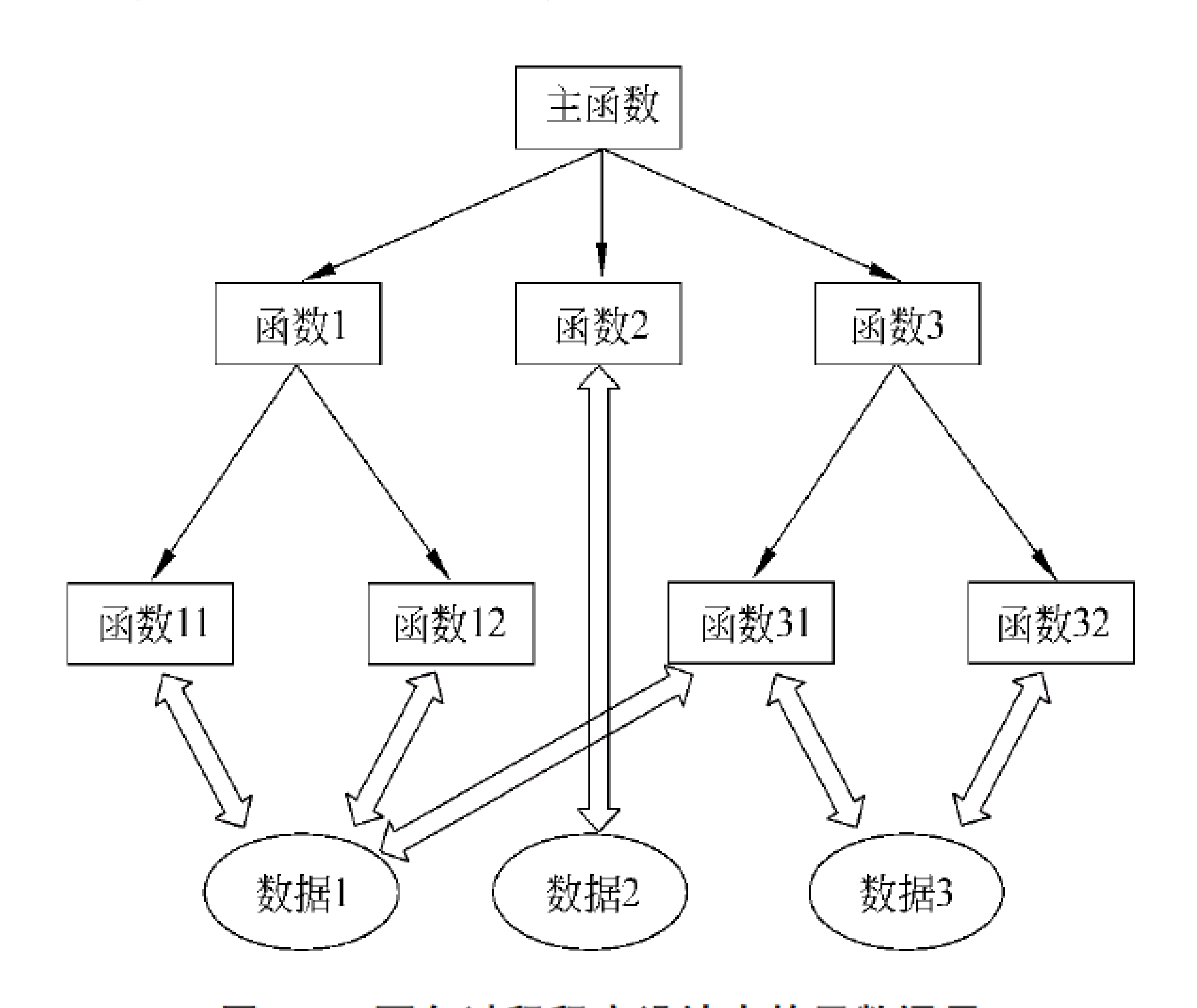


图 7-1 面向过程程序设计中的函数调用

当程序规模较大,数据很多、操作种类繁多时,程序设计者往往感到难以应付。就如工厂的厂长直接指挥每一个工人工作一样,一会儿让某车间的某工人在 A 机器上用 X 材

料生产轴承，一会儿又让另一个车间的某工人在B机器上用Y材料生产滚珠……，显然这是非常劳累的，而且往往会遗漏或搞错。

面向对象程序设计采取的是另外一种思路，它面对的是一个个对象。实际上，每一组数据都有特定的用途，是某种操作的对象，即一组操作调用一组数据。

例如，假设a、b、c是三角形的三条边，只与计算三角形面积和输出三角形的操作有关，与其他操作无关。我们就把这3个数据和对三角形的代码放在一起，封装成一个对象，与外界相对分隔。正如一个家庭的人生活在一起，与外界相对独立一样。

把数据与有关操作封装成一个对象，正如工厂把材料、机器和工人承包给车间，厂长只要向不同车间下达命令："一车间生产10台发动机"，"二车间生产100个轮胎"，"三车间生产15个车身"，……；车间就会运作起来，调动工人，选择有关材料，在不同的机器上完成相关的操作，把材料变成产品，厂长可以不必过问车间内工作的细节。对厂长来说，车间就如同一个"黑箱"，只要给它一个命令或通知，就能按规定完成任务。

面向对象程序设计者的任务包括两个方面：一是设计所需的各种类和对象，即确定哪些数据和操作封装在一起；二是考虑怎样向有关对象发送消息，以完成所需的任务。此时，如同有一个总调度，不断地向各个对象发出命令，让这些对象活动起来(或者说激活这些对象)，完成自己职责范围内的工作。当各个对象的操作都完成时，整体任务也就结束了，显然，对一个大型任务来说，面向对象程序设计方法是十分有效的，它能大大降低程序设计人员的工作难度，减少出错的机会。

7.2　类的定义和对象的创建

7.2.1　类和对象的关系

前面已说明什么是对象。每一个实体都可以作为对象。有些对象是具有相同的结构和特性的。例如高炮一连、高炮二连、高炮三连是三个不同的对象，但它们是属于同一个类型的，它们具有完全相同的结构和特性。而民兵一连、民兵二连、民兵三连这三个对象的类型也是相同的，但它们与高炮连的类型并不相同。每个对象都属于一个特定的类型。

在Python中对象的类型称为类(class)。类代表了某一批对象的共性和特性。类是对象的抽象，而对象是类的具体实例(instance)。要先定义类，然后才能用它去定义若干个同类型的对象。这就好比建造房屋先要设计图纸，然后按图纸在不同地方建造若干栋同类的房屋。

类和对象的关系如图7-2所示。

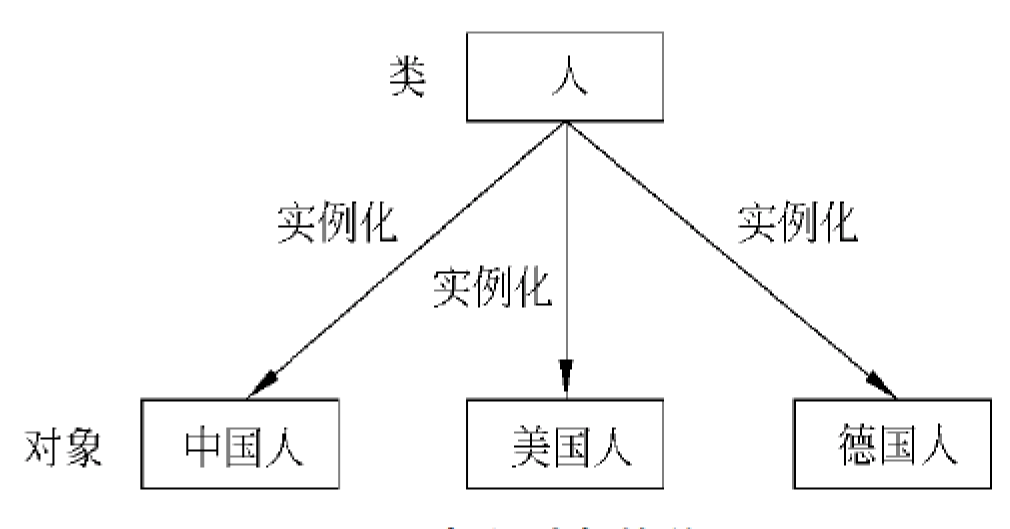


图7-2　类和对象的关系

注意：有的书中把对象称为实例，其实都是表述同一个意思。如无特别声明，本章所讲的对象指的是类的实例化对象。

7.2.2 类的定义

和其他的编程语言一样，Python 也是用 class 关键字来定义一个类的，在 Python 2.2 版本中，为了统一类(class)和类型(type)，引入了新式类。因此，Python 的类又可以分为经典类(旧式类)和新式类，它们的定义语法仅有很小的区别，如下所示：

经典类的定义：

```
class 类名：
    '类的文档字符串'
    类体
```

其中，第一行是类的文档字符串，用于说明该类是一个什么样的类，建议定义类时都写上。第二行是类体，由一系列的属性和方法(函数)组成。

新式类的定义：

```
class 类名 (父类):
    '类的文档字符串'
    类体
```

新式类和经典类的区别就是类名后要有括号，且指明其所继承的父类，默认是继承自 object 类，即 object 类是所有类的父类。如果有多个父类，需要用逗号分隔。关于继承的内容将本章后面介绍。

下面通过一个例子来说明如何定义一个类：

```
class A:                                #经典类 A
    'class A'                           #类的文档字符串
    version=1.0                         #类的属性
    def __init__(self,v):               #类的特殊方法 (构造函数)
        self.version=v                  #初始化对象属性
        print 'called __init__!'
    def printMessage(self):             #类的方法
        print 'called printMessage!'
        print A.version
```

说明：

上面定义了一个经典类 A，第 2 行类的文档字符串说明这是一个 A 类。第 3 行定义了一个初始值为 1.0 的类属性 version。第 4～6 行定义(重载)了一个特殊的方法，这个方法用于初始化操作，如“self. version＝v”，把传进来的参数 v 赋给对象的属性 version。实际上，__init__函数是一个内建函数，这里是重载了__init__函数。在后面会讲到类的常见内建函数。第 7～9 行定义了 printMessage 方法，输出类的属性 version。

注意：

(1) 类中定义的方法的参数列表至少有一个参数，通常把第一个参数指定为 self(也

可以是其他的标识符),self表示所创建的对象,在Java语言中相当于this。

(2) __init__方法在创建对象时由Python解析器自动调用,并且把创建对象时指定的所有参数(包括self参数)都传给__init__方法。

(3) 类的属性和对象的属性是不同的概念,在后面会举例说明。

7.2.3 对象的创建

对象是类的实例,对象的创建过程也可以说是类的实例化过程。在Java语言中,实例化一个对象需要使用New关键字,而在Python语言中则不需要使用new关键字。创建对象和调用函数类似,即类名(),如果__init__函数声明有参数,则还要传入相应的参数。否则会抛TypeError异常,提示参数不一致。创建对象后还要把它赋给一个变量,使该变量指向这个对象,否则将无法引用所创建的对象。

下面通过一个例子理解类的定义和对象的创建:

```
#coding:utf-8
#例 7-1  类的定义和对象的创建
class Person(object):
    '定义了一个 Person 类'
    #重载__init__方法
    def __init__(self,name,gender,age,nation):
        print '创建 Person 对象时调用__init__方法!'
        #初始化对象属性
        self.name=name
        self.gender=gender
        self.age=age
        self.nation=nation

    #定义设置对象 name 属性的方法
    def setName(self,name):
        self.name=name

    #定义设置对象 gender 属性的方法
    def setGender(self,gender):
        self.gender=gender

    #定义设置对象 age 属性的方法
    def setAge(self,age):
        self.age=age

    #定义设置对象 nation 属性的方法
    def setNation(self,nation):
        self.nation=nation

    #定义获取对象 name 属性的方法
```

```
        def get(self):
            return self.name

        #定义获取对象 gender 属性的方法
        def getGender(self):
            return self.gender

        #定义获取对象 age 属性的方法
        def getAge(self):
            return self.age

        #定义获取对象 nation 属性的方法
        def getNation(self):
            return self.nation

        def printMessage(self):
            print '姓名:%s,性别:%s,年龄:%d,国籍:%s' % (self.name,self.gender,self.age,
self.nation)

#创建一个姓名:小王,性别:男,年龄:26,国籍:中国的"中国人"
chinesePerson=Person('小王','男',26,'中国')
#创建一个姓名:Lucy,性别:女,年龄:23,国籍:美国的"美国人"
americanPerson=Person('Lucy','女',23,'美国')
#创建一个姓名:Pander,性别:男,年龄:21,国籍:德国的"德国人"
germanPerson=Person('Pander','男',21,'德国')
#分别调用上面三个对象的 printMessage 方法
print 'chinesePerson 对象信息:'chinesePerson.printMessage()
print 'americanPerson 对象信息:'americanPerson.printMessage()
print 'germanPerson 对象信息:'germanPerson.printMessage()
#通过函数的方式设置和获取 chinesePerson 对象的 age 属性
chinesePerson.setAge(27)
print '当前 chinesePerson 对象 age 属性的值为:',chinesePerson.getAge()
#可以直接设置和获取 chinesePerson 对象的 age 属性
chinesePerson.age=28
print '当前 chinesePerson 对象 age 属性的值为:',chinesePerson.age
```

程序运行结果如下：

```
创建 Person 对象时调用__init__方法!
创建 Person 对象时调用__init__方法!
创建 Person 对象时调用__init__方法!
chinesePerson 对象信息:
姓名:小王,性别:男,年龄:26,国籍:中国
americanPerson 对象信息:
姓名:Lucy,性别:女,年龄:23,国籍:美国
```

```
germanPerson 对象信息:
姓名:Pander,性别:男,年龄:21,国籍:德国
当前 chinesePerson 对象 age 属性的值为: 27
当前 chinesePerson 对象 age 属性的值为: 28
```

该程序定义了一个 Person,类中重载了__init__方法,用于初始化对象的 name、gender、age、nation 这 4 个属性。还定义了 getX 和 setX(X 代表属性名)方法分别获取和设置对象的 4 个属性。最后定义了 printMessage 方法,输出对象的信息。程序中创建了三个对象 chinesePerson、americanPerson、germanPerson,分别代表“中国人”、“美国人”、“德国人”。创建时指定了相应的参数。创建对象时会自动调用__init__方法,并且给定的参数都会传给该方法。调用类的方法需要用点操作符的形式指明调用哪个对象的方法,如 chinesePerson.printMessage()。获取对象的属性可以用点操作符的形式直接获取,如 chinesePerson.age。设置对象的属性可以通过赋值语句,如 chinesePerson.age=28。

7.3　类、对象的属性和方法

前面已指出,类是由属性和方法组成的。其中属性是对数据的封装,而方法则是对象所具有的行为(功能)。但属性和方法又因其是属于类的还是对象的而表现出不同的特性。此外,属性和方法又可分为共有和私有,在 C++ 和 Java 语言中,对属性和方法的公有和私有都是通过访问修饰符来区分的,例如公有属性和私有属性分别使用访问修饰符 public 和 private。但在 Python 中,由于没有这些访问修饰符(关键字),所以 Python 中属性和方法的共有和私有是通过标识符的约定来区分的。下面将对类、对象的属性和方法做详细的介绍。

7.3.1　属性

1. 属性根据所属的对象可以分为类属性和对象属性

在类内,且在方法外定义的,无特别声明的变量称为类属性,或者称为静态属性。在 C++ 和 Java 语言中,静态变量(属性)用 static 关键字声明。类属性既可以通过类名来访问,又可以通过对象名来访问。对象属性要放在方法中声明,且有对象名(通常为 self).前缀,只能通过对象名访问。

下面通过一个例子来理解类属性和对象属性:

```
#coding:utf-8
#例 7-2  类属性和对象属性
class Person(object):
    '定义了一个 Person 类'
    #定义类属性并赋初值
    nation='中国'
    city='厦门'
    def __init__(self,name,age):
        #定义对象属性并根据参数设置其值
```

```
        self.name=name
        self.age=age

p1=Person('小王',26)
p2=Person('小曾',24)
print '通过类访问类的属性 nation:%s,city:%s' % (Person.nation,Person.city)
print '通过对象 p1 访问类的属性 nation:%s,city:%s' % (p1.nation,p1.city)
print '通过对象 p2 访问类的属性 nation:%s,city:%s' % (p2.nation,p2.city)
print '通过对象 p1 访问它的对象属性 name:%s,age:%d' % (p1.name,p1.age)
print '通过对象 p2 访问它的对象属性 name:%s,age:%d' % (p2.name,p2.age)

#为对象 p1、p2 增加属性 city(对象属性)
print '为对象 p1、p2 增加属性 city(对象属性)后'
p1.city='清远'
p2.city='深圳'
print '通过类访问类的属性 nation:%s,city:%s' % (Person.nation,Person.city)
print '通过对象 p1 访问类的属性 nation:%s,city:%s' % (p1.nation,p1.city)
print '通过对象 p2 访问类的属性 nation:%s,city:%s' % (p2.nation,p2.city)
print '试图通过类 Person 访问对象的属性 name:%s,age:%d' % (Person.name,Person.age)
```

程序运行结果如下：

```
通过类访问类的属性 nation:中国,city:厦门
通过对象 p1 访问类的属性 nation:中国,city:厦门
通过对象 p2 访问类的属性 nation:中国,city:厦门
通过对象 p1 访问它的对象属性 name:小王,age:26
通过对象 p2 访问它的对象属性 name:小曾,age:24
为对象 p1、p2 增加属性 city(对象属性)后
通过类访问类的属性 nation:中国,city:厦门
通过对象 p1 访问类的属性 nation:中国,city:清远
通过对象 p2 访问类的属性 nation:中国,city:深圳

Traceback (most recent call last):
  File "C:/Python27/7.2.py", line 26, in<module>
    print '试图通过类 Person 访问对象的属性 name:%s,age:%d' % (Person.name,Person.age)
AttributeError: type object 'Person' has no attribute 'name'
```

该程序定义了一个 Person 类，类中定义了两个类属性 nation 和 city，并初始化为中国和厦门。然后在__init__方法中定义了两个对象属性 name 和 age。然后在主函数中创建了两个对象 p1 和 p2。可以看到，通过类和通过对象都可以访问类的属性 nation 和 city。输出都是中国和厦门。通过对象可以访问到它们各自的对象属性。读者可能已经发现，在为对象 p1、p2 增加对象属性 city，并分别赋值为清远和深圳后，再次通过对象输出属性 city(输出的实际上的对象属性)时，已经不再是厦门，而分别是清远和深圳。这是因为通过对象访问属性时，Python 解析器首先查找对象是否有指定的对象属性，如果有，

则停止查找，并返回其值，如果没有，就会查找是否有指定的类属性。如果还没有找到，则会抛 AttributeError 异常，提示没有指定的属性。类的属性被同名的对象属性“屏蔽”了，当然，可以通过对象名.__class__.属性名来访问被“屏蔽”的类属性。最后一条语句试图通过类 Person 访问对象的属性，很显然，这是不行的。

2. 属性根据访问的权限可以分为公有属性和私有属性

在 C++ 和 Java 语言中，公有属性和私有属性分别使用访问修饰符 public 和 private 声明，而在 Python 中是通过标识符的约定来区分的。如果属性的标识符(名称)以两个下划线开头，则说明是私有属性，否则是公有属性。

注意：公有属性和前面例子中的访问一样，而私有属性则通过如下语法才能够访问：

```
类(对象)名._类名__私有属性名
```

其中，类名前是一个下划线，类名后是两个下划线。

下面通过一个例子来理解私有属性和公有属性：

```
#coding:utf-8
#例 7-3　私有属性和公有属性
class Car(object):
    '定义了一个 Car 类'
    salesPrice=150000                #公有类属性
    __manufacturePrice=120000        #私有类属性
    def __init__(self,brand,serial):
        self.brand=brand             #公有对象属性
        self.__serial=serial         #私有对象属性

print '访问类的公有属性 salesPrice:',Car.salesPrice
print '访问类的私有属性 manufacturePrice:',Car.__manufacturePrice
c=Car('大众','一汽高尔夫')
print '访问对象 c 的公有属性 brand:',c.brand
print '访问对象 c 的私有属性 serial:',c.__serial
```

程序运行结果如下：

```
访问类的公有属性 salesPrice: 150000
访问类的私有属性 manufacturePrice:


Traceback (most recent call last):
  File "C:/Python27/7.3.py", line 10, in<module>
    print '访问类的私有属性 manufacturePrice:',Car.__manufacturePrice
AttributeError: type object 'Car' has no attribute '__manufacturePrice'
```

报错原因是因为访问类的私有属性的方式不正确，把 Car.__manufacturePrice，C.__serial 改为 Car._Car__manufacturePrice，c._Car__serial 后程序的运行结果如下：

```
访问类的公有属性 salesPrice: 150000
```

```
访问类的私有属性 manufacturePrice: 120000
访问对象 c 的公有属性 brand: 大众
访问对象 c 的私有属性 serial: 一汽高尔夫
```

此外，还有内置属性，如__doc__、__bases__等，是由 Python 解析器提供的，用于管理类的内部关系。

类的常用的内置属性如表 7-1 所示。

表 7-1　类的常用的内置属性

内置属性名	说　　明
__dict__	类的属性组成的字典
__doc__	类的文档字符串
__module__	类定义所在的模块
__name__	类的名字(字符串)
__bases__	类的父类组成的元组

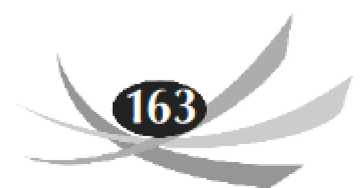
下面通过一个例子来理解类的内置属性。

```
#coding:utf-8
#例 7-4　类的内置属性
class BuiltAttribute(object):
    'class BuiltAttribute'
    pass

print '调用内置函数 dir 求类的内置属性和方法:'
print dir(BuiltAttribute)
print 'BuiltAttribute 类的__dict__属性:',BuiltAttribute.__dict__
print 'BuiltAttribute 类的__doc__属性:',BuiltAttribute.__doc__
print 'BuiltAttribute 类的__module__属性:',BuiltAttribute.__module__
print 'BuiltAttribute 类的__name__属性:',BuiltAttribute.__name__
print 'BuiltAttribute 类的__bases__属性:',BuiltAttribute.__bases__
```

程序运行结果如下：

```
调用内置函数 dir 求类的内置属性和方法:
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__
weakref__']
BuiltAttribute 类的__dict__属性: {'__dict__':<attribute '__dict__' of
'BuiltAttribute' objects>, '__module__': '__main__', '__weakref__':<attribute '__weakref
__' of 'BuiltAttribute' objects>, '__doc__': 'class BuiltAttribute'}
BuiltAttribute 类的__doc__属性: class BuiltAttribute
BuiltAttribute 类的__module__属性: __main__
```




```
BuiltAttribute类的__name__属性：BuiltAttribute
BuiltAttribute类的__bases__属性：(<type 'object'>,)
```

7.3.2　方法

通过对属性的学习，我们已经知道属性可以分为公有属性和私有属性、类属性和对象属性。同样的，方法也可以分为公有方法和私有方法、类方法和对象方法。此外，还有静态方法。

1. 公有方法和私有方法

定义公有方法无须特别声明，而定义私有方法时，方法名要以两个下划线开头。调用时，无论是公有方法还是私有方法，都可以通过类或者对象的方式调用。但是如果通过类的方式调用，则必须要传入一个对象。且调用私有方法还必须通过以下语法调用：

```
类(对象)名._类名__私有方法名()
```

可以看到，这和访问私有属性非常相似，只是把私有属性名改成私有方法名（以函数的形式调用）。

下面通过一个例子来理解公有方法和私有方法。

```
#coding:utf-8
#例 7-5　公有方法和私有方法
class Methods(object):
    '定义一个 Methods 类'
    #定义公有方法
    def publicMethod(self):
        return '公有方法 publicMethod!'

    #定义私有方法
    def __privateMethod(self):
        return '私有方法 privateMethod!'

m=Methods()
print '以对象的方式调用',m.publicMethod()
#以类的方式调用公有方法 publicMethod 传入了一个对象 m
print '以类的方式调用',Methods.publicMethod(m)
print '以对象的方式调用',m._Methods__privateMethod()
print '以类的方式调用',Methods._Methods__privateMethod()
```

程序运行结果如下：

```
以对象的方式调用 公有方法 publicMethod!
以类的方式调用 公有方法 publicMethod!
以对象的方式调用 私有方法 privateMethod!
以类的方式调用
```

```
Traceback (most recent call last):
  File "C:/Python27/7.5.py", line 17, in<module>
    print '以类的方式调用',Methods._Methods__privateMethod()
TypeError: unbound method __privateMethod() must be called with Methods instance
as first argument (got nothing instead)
```

可以看到，以类的方式调用公有方法时传入了一个对象m，可以正常执行。第4条调用语句，以类的方式调用私有方法时没有传入类的实例化对象，所以抛TypeError异常，提示未绑定的方法(以对象的方式调用的方法称为绑定的方法)必须以类的实例化对象作为第一个参数。第4条调用语句传入一个类的实例化对象m后，程序的运行结果如下：

```
以对象的方式调用 公有方法 publicMethod!
以类的方式调用 公有方法 publicMethod!
以对象的方式调用 私有方法 privateMethod!
以类的方式调用 私有方法 privateMethod!
```

注意：上面介绍的公有方法和私有方法都是对象的方法，关于类的公有方法和私有方法在下面的类方法和静态方法所举的例子中有叙述。

2. 类方法和静态方法

定义类方法时可以通过@classmethod指令的方式定义或者通过使用内建函数classmethod的方式将一个普通的方法转为类方法。类似的，定义静态方法时可以通过@staticmethod指令的方式定义或者通过使用内建函数staticmethod的方式将一个普通的方法转为静态方法。调用时，无论是类方法还是静态方法，都可以通过类或者对象的方式调用。

下面通过一个例子来理解类方法和静态方法。

```
#coding:utf-8
#例 7-6  类方法和静态方法
class Methods(object):
    '定义一个 Methods 类'
    #通过@classmethod指令定义公有类方法
    @classmethod
    def publicClassMethod(cls):
        return cls

    #通过@classmethod指令定义私有类方法
    @classmethod
    def __privateClassMethod(cls):
        return cls

    #通过@staticmethod指令定义公有静态方法
    @staticmethod
    def publicStaticMethod():
        return '公有静态方法 publicStaticMethod!'
```

```
    #通过@staticmethod指令定义私有静态方法
    @staticmethod
    def __privateStaticMethod():
        return '私有静态方法 privateStaticMethod!'

    #定义公有方法
    def publicMethod(self):
        print 'called publicMethod!'

    #定义私有方法
    def __privateMethod(self):
        print 'called privateMethod!'

    #通过内建函数 classmethod将公有方法 publicMethod转为类方法
    publicMethodToClassMethod=classmethod(publicMethod)

    #通过内建函数 classmethod将私有方法 privateMethod转为类方法
    privateMethodToClassMethod=classmethod(__privateMethod)

    #通过内建函数 staticmethod将公有方法 publicMethod转为静态方法
    publicMethodToStaticMethod=staticmethod(publicMethod)

    #通过内建函数 staticmethod将私有方法 privateMethod转为静态方法
    privateMethodToStaticMethod=staticmethod(__privateMethod)

m=Methods()
print '以类的方式调用通过@classmethod指令定义的公有类方法
publicClassMethod!',Methods.publicClassMethod()
print '以对象的方式调用通过@classmethod指令定义的公有类方法
publicClassMethod!',m.publicClassMethod()
print '以类的方式调用通过@classmethod指令定义的私有类方法
privateClassMethod!',Methods._Methods__privateClassMethod()
print '以对象的方式调用通过@classmethod指令定义的私有类方法
privateClassMethod!',m._Methods__privateClassMethod()

print '以类的方式调用通过@staticmethod指令定义的',Methods.publicStaticMethod()
print '以对象的方式调用通过@staticmethod指令定义的',m.publicStaticMethod()
print '以类的方式调用通过@staticmethod指令定义的
',Methods._Methods__privateStaticMethod()
print '以对象的方式调用通过@staticmethod指令定义的',m._Methods__privateStaticMethod()

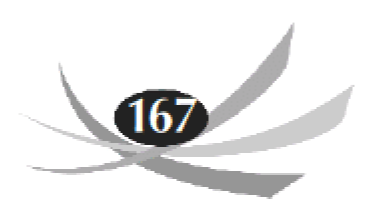
print '以类的方式调用通过内建函数 classmethod转换成的类方法
publicMethodToClassMethod!'
```

```
Methods.publicMethodToClassMethod()
print '以对象的方式调用通过内建函数 classmethod 转换成的类方法
publicMethodToClassMethod!'
m.publicMethodToClassMethod()
print '以类的方式调用通过内建函数 classmethod 转换成的类方法
privateMethodToClassMethod!'
Methods.privateMethodToClassMethod()
print '以对象的方式调用通过内建函数 classmethod 转换成的类方法
privateMethodToClassMethod!'
m.privateMethodToClassMethod()

print '以类的方式调用通过内建函数 staticmethod 转换成的类方法
publicMethodToStaticMethod!'
Methods.publicMethodToStaticMethod(m)
print '以对象的方式调用通过内建函数 staticmethod 转换成的类方法
publicMethodToStaticMethod!'
m.publicMethodToStaticMethod(m)
print '以类的方式调用通过内建函数 staticmethod 转换成的类方法
privateMethodToStaticMethod!'
Methods.privateMethodToStaticMethod(m)
print '以对象的方式调用通过内建函数 staticmethod 转换成的类方法
privateMethodToStaticMethod!'
m.privateMethodToStaticMethod(m)
```

程序运行结果如下：

```
以类的方式调用通过@classmethod 指令定义的公有类方法 publicClassMethod!<class
'__main__.Methods'>
以对象的方式调用通过@classmethod 指令定义的公有类方法 publicClassMethod!<class
'__main__.Methods'>
以类的方式调用通过@classmethod 指令定义的私有类方法 privateClassMethod!<class
'__main__.Methods'>
以对象的方式调用通过@classmethod 指令定义的私有类方法 privateClassMethod!<class '__
main__.Methods'>
以类的方式调用通过@staticmethod 指令定义的公有静态方法 publicStaticMethod!
以对象的方式调用通过@staticmethod 指令定义的公有静态方法 publicStaticMethod!
以类的方式调用通过@staticmethod 指令定义的私有静态方法 privateStaticMethod!
以对象的方式调用通过@staticmethod 指令定义的私有静态方法 privateStaticMethod!
以类的方式调用通过内建函数 classmethod 转换成的类方法 publicMethodToClassMethod!
called publicMethod!
以对象的方式调用通过内建函数 classmethod 转换成的类方法 publicMethodToClassMethod!
called publicMethod!
以类的方式调用通过内建函数 classmethod 转换成的类方法 privateMethodToClassMethod!
called privateMethod!
以对象的方式调用通过内建函数 classmethod 转换成的类方法 privateMethodToClassMethod!
```




```
called privateMethod!
以类的方式调用通过内建函数 staticmethod 转换成的类方法 publicMethodToStaticMethod!
called publicMethod!
以对象的方式调用通过内建函数 staticmethod 转换成的类方法 publicMethodToStaticMethod!
called publicMethod!
以类的方式调用通过内建函数 staticmethod 转换成的类方法 privateMethodToStaticMethod!
called privateMethod!
以对象的方式调用通过内建函数 staticmethod 转换成的类方法 privateMethodToStaticMethod!
called privateMethod!
```

3. 内置方法

类的方法除了前面介绍的对象方法、类方法、静态方法外，还有内置方法，内置方法中的一部分有默认的行为，而另一部分则没有，留到需要的时候去实现。这些内置方法是Python中用来扩展类的强有力的方式。表7-2列出了比较常用的内置方法。

表7-2　类的常用的内置方法

内置方法	说　明
__init__(self,...)	初始化对象，在创建新对象后调用
__del__(self)	释放对象，在对象被删除前调用
__new__(self, * args, * * key)	返回创建的对象，在创建对象时被调用
__str__(self)	在使用 print 语句时被调用
__delitem__(self,key)	从字典中删除 key 对应的元素
__setitem__(self,key,value)	为字典中的 key 赋值
__getitem__(self,key)	获取序列的索引 key 对应的值，等价于 seq[key]
__len__(self)	在调用内建函数 len()时被调用
__cmp__(src,dst)	比较两个对象 src 和 dst
__getattr__(self,attr)	获取属性的值
__settattr__(self,attr,val)	设置属性的值
__delattr__(self,attr)	删除 attr 属性
__gt__(self,other)	判断 self 对象是否大于 other 对象
__lt__(self,other)	判断 self 对象是否小于 other 对象
__ge__(self,other)	判断 self 对象是否大于或等于 other 对象
__le__(self,other)	判断 self 对象是否小于或等于 other 对象
__eq__(self,other)	判断 self 对象是否等于 other 对象
__call__(self, * args)	把实例对象作为函数调用

下面通过一个例子介绍其中的几个内置方法。

```
#coding:utf-8
```

```
#例 7-7  内置方法
class BuiltinMethods:
    '定义了一个 BuiltinMethods 类'
    datadict={}
    def __init__(self,name):
        self.name=name
        print '调用__init__方法'

    def printAttribute(self):
        print 'name:',self.name

    def __del__(self):
        print '调用__del__方法'

    def __str__(self):
        print '调用__str__方法'
        return self.name

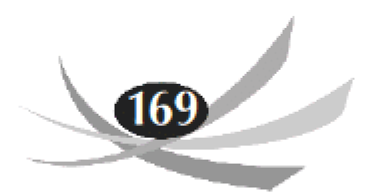
    def __setitem__(self,key,value):
        print '调用__setitem__方法'
        self.datadict[key]=value
        print '设置 key:%s,value:%s' % (key,value)

    def __getitem__(self,key):
        print '调用__getitem__方法'
        return self.datadict[key]

    def __delitem__(self,key):
        print '调用__delitem__方法'
        del self.datadict[key]
        print '删除了字典中的 PL1 项'

    def __getattr__(self,attr):
        print '调用__getattr__方法'

print '创建对象时'
bm=BuiltinMethods('Python')
bm.printAttribute()
print '为字典中的 PL1 赋值 C 时'
bm['PL1']='C'
print '为字典中的 PL2 赋值 Java 时'
bm['PL2']='Java'
print '获取字典中 PL1 的 value 值时'
value=bm['PL1']
```




```
print 'key为PL1对应的value值为:',value
print '删除字典中PL1项时'
del bm['PL1']
print '使用print语句时'
print bm
```

程序运行结果如下：

```
创建对象时
调用__init__方法
name: Python
为字典中的PL1赋值C时
调用__setitem__方法
设置key:PL1,value:C
为字典中的PL2赋值Java时
调用__setitem__方法
设置key:PL2,value:Java
获取字典中PL1的value值时
调用__getitem__方法
key为PL1对应的value值为: C
删除字典中PL1项时
调用__delitem__方法
删除了字典中的PL1项
使用print语句时
调用__str__方法
Python
```

可以看到这些内置方法都没有显式调用，都是在执行相应的语句时由Python隐式调用的。如执行bm['PL1']='C'语句时调用__setitem__方法，执行del bm['PL1']语句时调用__delitem__方法，执行print语句时调用__str__方法等。

7.4　组　　合

一个类被定义后，目的就是要把它当成一个模块来使用，并把它嵌入到我们的代码中。有两种方式可以利用这些已定义好的类，一种是组合，这种方式就是让不同的类混合并入其他的类，形成更加复杂、更符合需求的类，从而增加功能和提高代码重用性。另一种方式是通过继承。继承将在下一节介绍。

下面通过一个例子理解类的组合。

```
#coding:utf-8
#例7-8　类的组合
class Person(object):
    '定义了一个Person类'
    def __init__(self,name,age,gender,birthday,address,contact):
```

```
        #初始化 Person 类的实例化对象的属性
        self.name=name
        self.age=age
        self.gender=gender
        self.birthday=birthday
        self.address=address
        self.contact=contact

    def printBaseInfomation(self):
        print '基本信息:name:%s,age:%d,gender:%s' % (self.name,self.age,self.
        gender)

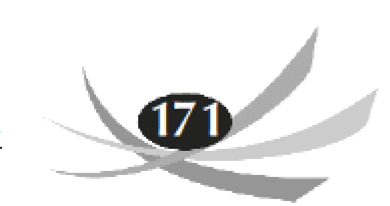
    def printBrithdayInfomation(self):
        print '生日:%d:%d:%d' % (self.birthday.year,self.birthday.month,self.
        birthday.day)

    def printAllInfomation(self):
        #调用 Person 类中定义的方法
        self.printBaseInfomation()
        #调用 Birthday 类中定义的方法
        self.printBrithdayInfomation()
        #调用 Address 类中定义的方法
        self.address.printAddressInfomation()
        #调用 Contact 类中定义的方法
        self.contact.printContactInfomation()

class Birthday(object):
    '定义了一个 Brithday 类'
    def __init__(self,year,month,day):
        self.year=year
        self.month=month
        self.day=day

class Address(object):
    '定义了一个 Address 类'
    def __init__(self,nation,province,city,region,street):
        self.nation=nation
        self.province=province
        self.city=city
        self.region=region
        self.street=street

    def printAddressInfomation(self):
        print '地址:%s,%s,%s,%s,%s' % (self.nation,self.province,self.city,self.
```




```
            region,self.street)

class Contact(object):
    '定义了一个 Contact 类'
    def __init__(self,telephone,wechat,qq):
        self.telephone=telephone
        self.wechat=wechat
        self.qq=qq

    def printContactInfomation(self):
        print '联系方式:手机号码:%s 微信:%s QQ:%s' % (self.telephone,self.wechat,
        self.qq)

#先分别创建 Birthday、Address、Contact 的实例化对象
birthday=Birthday(2000,1,1)
address=Address('中国','福建','厦门','思明','滨海街道')
contact=Contact('15912345678','xiaowuwechat','123456')
''' Person 类的实例化对象 person 的 birthday、address、contact 属性分别是
    Birthday 类的实例化对象 birthday
    Address 类的实例化对象 address
    Contact 类的实例化对象 contact
'''
#用上面创建的实例化对象 birthday、address、contact 创建 Person 类的实例化对象
person=Person('小吴',16,'女',birthday,address,contact)
person.printAllInfomation()
```

程序的运行结果如下：

```
基本信息:name:小吴,age:16,gender:女
生日:2000:1:1
地址:中国,福建,厦门,思明,滨海街道
联系方式:手机号码:15912345678 微信:xiaowuwechat QQ:123456
```

该程序定义了一个 Person 主类和三个从类 Birthday、Address、Contact，这个 Person 主类有 name、age、gender、birthday、address、contact 等对象属性。其中，name、age、gender 属性的类型是基本类型，而 birthday、address、contact 属性的类型则分别是 Birthday、Address、Contact 这三个类的实例化对象。此外，Person 类还定义了 4 个方法 __init__、printBaseInformation、printBirthdayInformation、printAllInformation。__init__ 方法用于初始化 Person 类实例化对象的属性信息；printBaseInformation 方法用于输出基本类型的属性信息，这个方法直接引用属性即可，如 self. name；printBirthdayInformation 方法用于输出 Birthday 类的实例化对象的属性信息（生日信息），这个方法必须要先获得 Birthday 的实例化对象（通过 self. birthday 获得），然后再获取它的生日信息，如 self. birthday. year。如果它的生日信息还是一个类的实例化对象，同样先获得这个类的实例化对象，这样一层层地通过属性点操作符最终获取所需要的信息；

printAllInformation 方法则用于输出 Person 类的所有信息，包括 Birthday 类的生日信息，Address 类的地址信息和 contact 类的联系方式信息，不同的是，这个方法是通过调用其他方法的方式实现的，而且调用又分为直接调用和间接调用，如调用主类 Person 定义的 printBaseInformation、printBirthdayInformation 方法是直接调用(self. printBaseInfomation())，而调用 Address 类的 printAddressInfomation 方法和 Contact 类的 printContactInfomation 方法是间接调用(self. address. printAddressInfomation()、self. contact. printContactInfomation())。Birthday 类有三个对象属性 year、month 和 day。Address 类有 5 个对象属性 nation、province、city、region 和 street。而 Contact 类有三个对象属性 telephone、wechat 和 qq。显然，这三个类和 Person 类属于组合关系，可以说这三个简单的类(从类)组合成了一个复杂的类(主类)。在主函数中，首先实例化这三个类，然后在实例化 Person 类的过程中使用了这三个类的实例化对象。最后调用 person 对象的 printAllInfomation 方法输出所有的信息。

7.5 继承与派生

面向对象的程序设计有三个主要特点：封装、继承和多态性。在前面我们学习了类和对象，了解了面向对象程序设计的其中一个重要特征——封装，已经能够设计出基于对象的程序，这是面向对象程序设计的基础。在本节中主要介绍有关继承的知识。

继承性是面向对象程序设计最重要的特征，可以说，如果没有掌握继承性，就等于没有掌握类和对象的精华，就是没有掌握面向对象程序设计的真谛。

在传统的程序设计中，人们往往要为每一种应用项目单独地进行一次程序开发，因为每一种应用有不同的目的和要求，程序的结构和具体的编码是不同的，人们无法使用已有的软件资源。即使两种应用具有许多相同或相似的特点，程序设计者可以吸取已有程序的思路，作为自己软件开发新程序的参考，但是人们仍然不得不另起炉灶，重写程序或者对已有的程序进行较大的改写。显然，这种方法的重复工作量是很大的，这是因为过去的程序设计方法和计算机语言缺乏软件重用的机制。人们无法利用现有的丰富的软件资源，这就造成软件开发过程中人力、物力和时间的巨大浪费，效率较低。

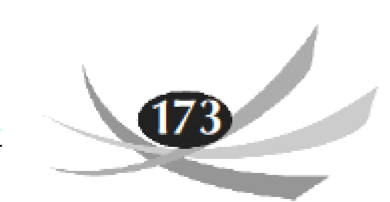
面向对象技术强调软件的可重用性(software reusability)。Python 语言提供了类的继承机制，解决了软件重用问题。

7.5.1 继承与派生的概念

前面已经说过，函数可以提高代码的重用性，但这种代码重用性比较有限，要想实现代码更高的重用性，可以通过继承(inheritance)这一机制实现。因此，继承是 Python 面向对象程序设计中一个重要的组成部分。

前面介绍了类，一个类中包含了若干的属性(对象属性居多)和方法。在不同的类中，属性和方法是不同的，但有时两个类的内容基本相同或有一部分相同。例如定义了学生基本数据的类 Student：

```
class Student(object):
```




```
    '定义了一个 Student类'
    def __init__(self,num,name,gender):
        self.num=num
        self.name=name
        self.gender=gender

    def printInformation(self):
        print 'num:%d,name:%s,gender:%s' \
    %(self.num,self.name,self.gender)
```

如果学校的某一部门除了需要用到学号、姓名、性别以外,还需要用到年龄、地址等信息。当然可以重新定义另一个类 Student1:

```
class Student1(object):
    '定义了一个 Student1类'
    def __init__(self,num,name,gender,age,address):          #此行很相似
        self.num=num                                          #此行已有
        self.name=name                                        #此行已有
        self.gender=gender                                    #此行已有
        self.age=age
        self.address=address

    def printInformation(self):                               #此行已有
        print 'num:%d,name:%s,gender:%s,age:%d,address:%s' \  #此行很相似
        %(self.num,self.name,self.gender,self.age,self.address) #此行很相似
```

可以看到有相当一部分是原来已有的。很多人自然会想到能否利用原来定义的类 Student 作为基础,再加上新的内容即可,以减少重复的工作量。Python 提供的继承机制就是为了解决此类问题。

在本章开头已举了马的例子来说明继承的概念。“公马”继承了“马”的全部特征,再加上“雄性”的新特征。“白公马”继承了“公马”的全部特征,再增加“白色”的特征。“公马”是“马”派生出来的一个分支,“白公马”是“公马”派生出来的一个分支。如图 7-3 所示。

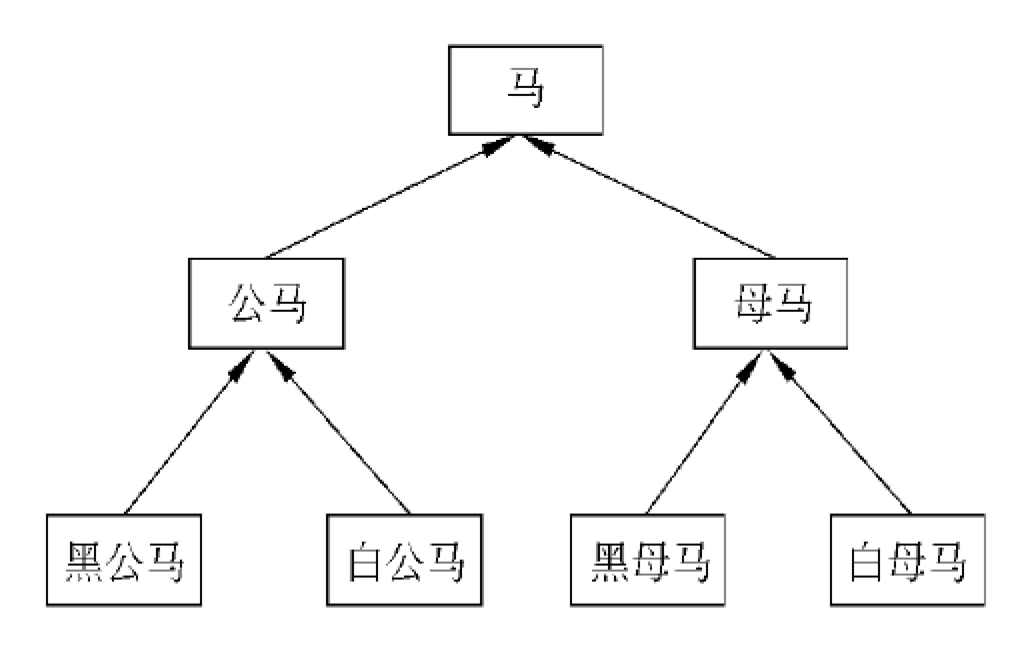


图 7-3　马、公马、白公马的继承关系

所谓“继承”就是在一个已存在的类的基础上建立一个新的类。已存在的类(例如“马”)

称为“基类”或“父类”。新建立的类(例如“公马”)称为“派生类”或“子类”。如图 7-4 所示。

一个新类从已有的类那里获得其已有特性,这种现象称为类的继承。通过继承,一个新建子类从已有的父类那里获得父类的特性。从另一个角度说,从已有的类(父类)产生一个新的子类,称为类的派生。类的继承是用已有的类来建立专用类的编程技术。派生类继承了基类的所有属性和方法,并可以对属性和方法做必要的增加和调整。一个基类可以派生出多个派生类,每个派生类又可以作为基类再派生出新的派生类,因此基类和派生类是相对而言的。一代一代地派生下去,就形成类的继承层次结构。相当于一个大的家族,有许多分支,所有的子孙后代都继承了祖辈的基本特征,同时又有区别和发展。类的每一次派生,都继承了其基类的基本特征,同时又根据需要调整和扩充原有的特征。

以上介绍的是最简单的情况:一个派生类只继承自一个类,这称为单继承(single inheritance),这种继承关系所形成的层次是一个树状结构,可以用图 7-5 表示。

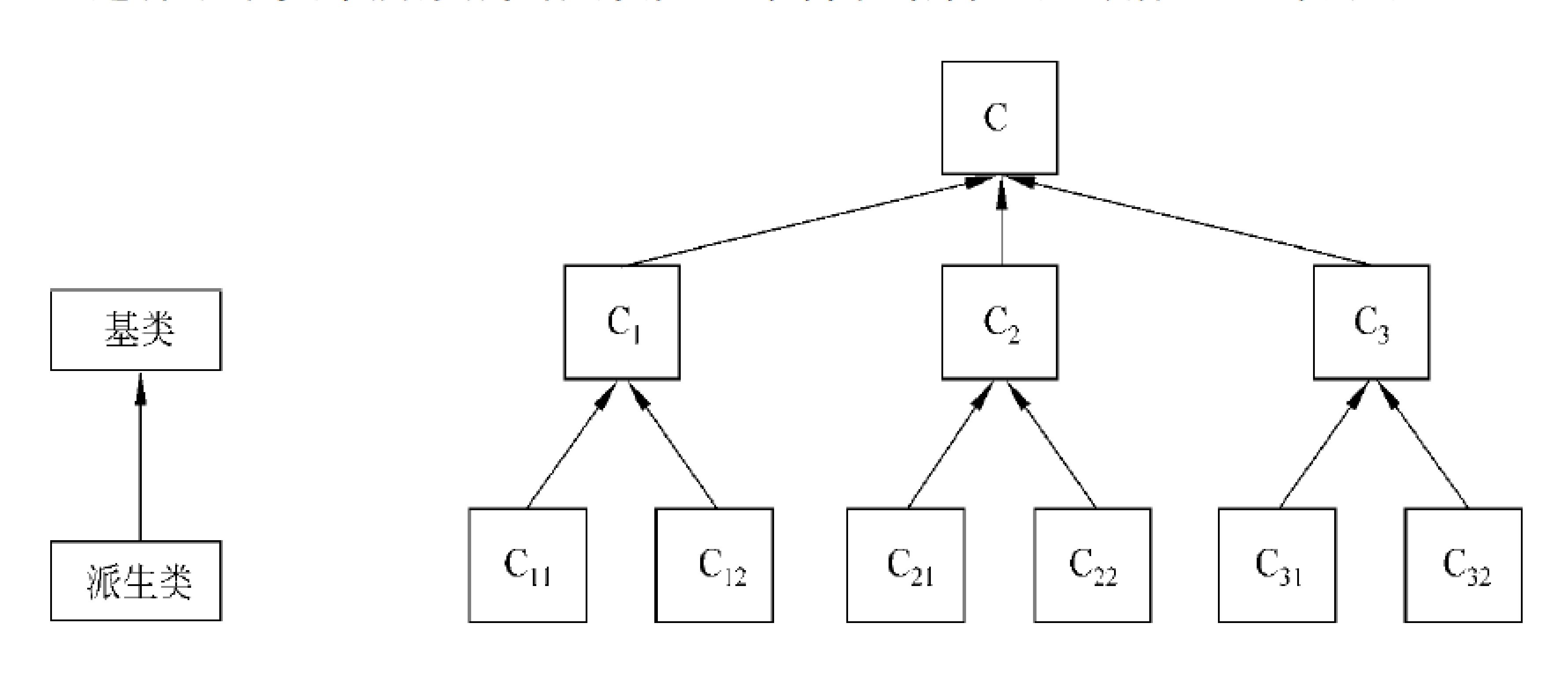


图 7-4 基类和派生类的继承示意图

图 7-5 单继承的树状结构

请注意图中的箭头方向,在本章中约定,箭头表示继承的方向,从派生类指向基类。

一个派生类不仅可以从一个基类派生,也可以从多个基类派生,也就是说,一个派生类可以有两个或多个基类(或者说,一个子类可以有两个或多个父类)。例如马和驴杂交所生下的骡子就有两个基类——马和驴。骡子既继承了马的一些特征,也继承了驴的一些特征。又如“计算机专科”,是从“计算机专业”和“大专层次”派生出来的子类,它具备两个基类的特征。一个派生类有两个或多个基类的称为多重继承(multiple inheritance),这种继承关系形成的结构如图 7-6 所示。

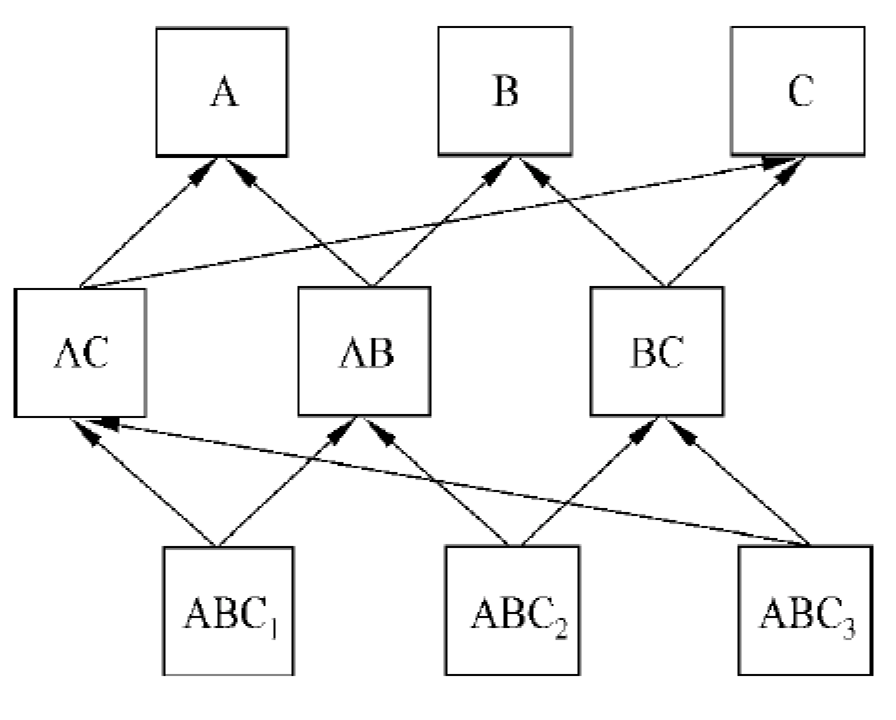
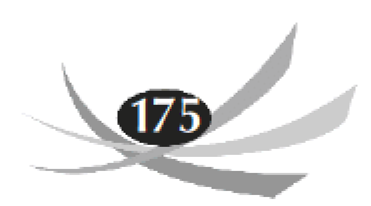


图 7-6 多继承的网状结构

关于基类和派生类的关系,可以表述为:派生类是基类的具体化,而基类则是派生类的抽象。从图 7-7 中可以看到:小学生、中学生、大学生、研究生、留学生是学生的具体化,他们是在学生的共性基础上加上某些特点形成的子类。而学生则是对各类学生共性的综合,是对各类具体学生特点的抽象。基类综合了派生类的公共特征,派生类则在基类的基础上增加某些特性,把抽象类变成具体的、



实用的类型。

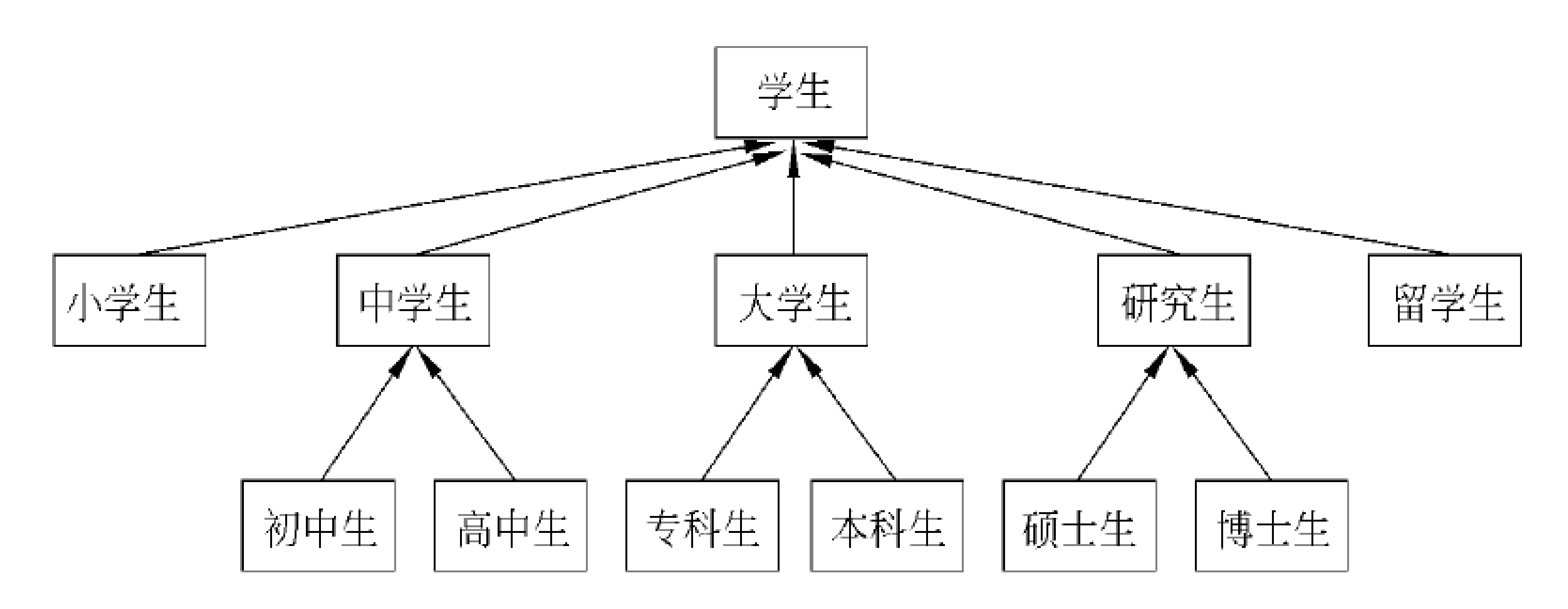


图7-7　继承与派生的关系图

7.5.2　派生类的定义

定义派生类的一般形式：

```
class 派生类类名(基类类名):
    def __init__(self [,args]):                    #构造方法
        基类类名.__init__(self [,args])            #调用基类的构造方法
        [新增属性的赋值]
```

在派生类类名后的括号内指定所要继承的基类类名，由于定义一个类通常都会包含__init__构造方法，所以在派生类中首先也会定义该方法，在该方法中先调用基类的构造方法，并传以必要的参数，用于初始化基类的属性。然后会通过赋值语句初始化派生类中新增加的属性。

先通过一个例子说明怎样通过继承来建立派生类，从最简单的单继承开始：

```
coding:utf-8
#例7-9　派生类的定义
class Student(object):
    '定义了一个Student类'
    baseClassData=123
    def __init__(self,num,name,gender):
        self.num=num
        self.name=name
        self.gender=gender

    def printInformation(self):
        print 'num:%d,name:%s,gender:%s'   (self.num,self.name,self.gender)

class Student1(Student):                               #声明基类是Student
    '定义了一个Student1类'
    def __init__(self,num,name,gender,age,address):
                                                 #比基类的__init__方法多传两个参数
        Student.__init__(self,num,name,gender)         #调用基类的__init__方法
```

```
        self.age=age                                #新增加的对象属性
        self.address=address                        #新增加的对象属性

    def printInformation1(self):
        Student.printInformation(self)              #调用基类的 printInformation 方法
        print 'age:%d,address:%s' % (self.age,self.address)
                                                    #输出新增加的对象属性

#创建 Student1 类的实例化对象
s=Student1(123456,'小王','男',26,'广东')
s.printInformation1()
#通过派生类的实例化对象调用基类的类属性 baseClassData
print '通过派生类的实例化对象调用从基类继承过来的类属性 baseClassData:',
s.baseClassData
```

在定义 Student1 类时,括号内的 Student 就是所继承的基类,Student1 类就是 Student 类的派生类。其实,在前面介绍的例子中绝大多数都是以继承的方式创建的类(派生类)。只是继承的基类不是用户定义的类,而是编译器自带的 object 类。在这个 Student1 的类中,定义了__init__构造方法,其参数列表比基类中__init__构造方法多了两个参数,它们用于初始化派生类中新增加的对象属性(age,address)。方法内首先调用基类的__init__方法,初始化基类的对象属性。然后再通过赋值语句初始化派生类中新增加的对象属性。接着又定义了 printInformation1 方法,该方法首先调用基类的 printInformation 方法,输出基类的对象属性,然后再输出派生类新增加的对象属性。可以看到这种通过继承定义的派生类比重新定义满足要求的类更简洁,如果类更复杂,简洁性就更明显。在程序的主函数中首先创建 Student1 类的实例化对象,然后调用派生类的 printInformation1 方法,最后通过派生类的实例化对象调用基类的类属性 baseClassData,程序的运行结果如下:

```
num:123456,name:小王,gender:男
age:26,address:广东
通过派生类的实例化对象调用基类的类属性 baseClassData: 123
```

说到继承,必然会提到如何调用基类中的方法,一般使用非绑定的类方法,即通过类名访问基类中的方法,并在参数列表中引入对象 self,从而达到调用基类方法的目的。如上面这个例子中的 Student.__init__(self,num,name,gender)和 Student.printInformation(self)都是通过这种方式调用基类的方法。但这种方式有一个弊端,就是当基类的类名改动或者派生类改为继承其他的类时,在派生类中通过类名调用基类方法的所有地方中的类名都需要修改成改动后的类名,如果是简短的代码,这样的改动还可以接受,但如果代码量很大,改动的工作量也是很大的。为了解决这个问题,Python 增加了 super 内建函数来调用基类中的方法。把上面例子中的 Student.__init__(self,num,name,gender)和 Student.printInformation(self)分别改为 super(Student1,self).__init__(num,name,gender)和 super(Student1,self).printInformation()。运行结果不变。

现在假设基类 Student 的类名改为 CollegeStudent，我们只需改动继承的基类类名一个地方即可。

使用 super 内建函数来调用基类中的方法，如果基类的名称改变或者派生类改为继承其他的类时，只需要修改派生类继承基类的名称即可。这样既可以将代码的维护量降到最低，又可以提高程序开发的周期。因此，定义派生类更一般的形式为：

```
class 派生类类名(基类类名):
    def __init__(self [,args]):                        #构造方法
        super(派生类类名,self).__init__([args])         #调用基类的构造方法
        [新增属性的赋值]
```

但是，有一种特殊情况，当派生类继承两个或多个基类时，且这两个或多个基类有同名的方法(如__init__)，通过 super 内建函数并不能智能地分辨出调用哪个基类的方法，如果它们的参数个数相同，这会导致某些基类的方法被调用多次，而某些基类的方法一次都没被调用。如果不同，程序会抛 TypeError 异常，提示所需参数和给定参数不一致。这种情况将在 7.5.4 节多重继承中举例说明。

7.5.3　派生类的组成

派生类中的属性和方法包括从基类继承过来的属性和方法以及自己增加的属性和方法两大部分。从基类继承的属性和方法体现了派生类从基类继承而获得的共性，而新增加的属性和方法体现了派生类的个性。正是这些新增加的属性和方法体现了派生类和基类的不同，体现了不同派生类之间的区别。图 7-8 可以形象地表示继承关系。在基类中包括属性和方法两部分，派生类分为两大部分：一部分是从基类继承来的属性和方法，另一部分是在定义派生类时增加的部分。每一部分均分别包括属性和方法。

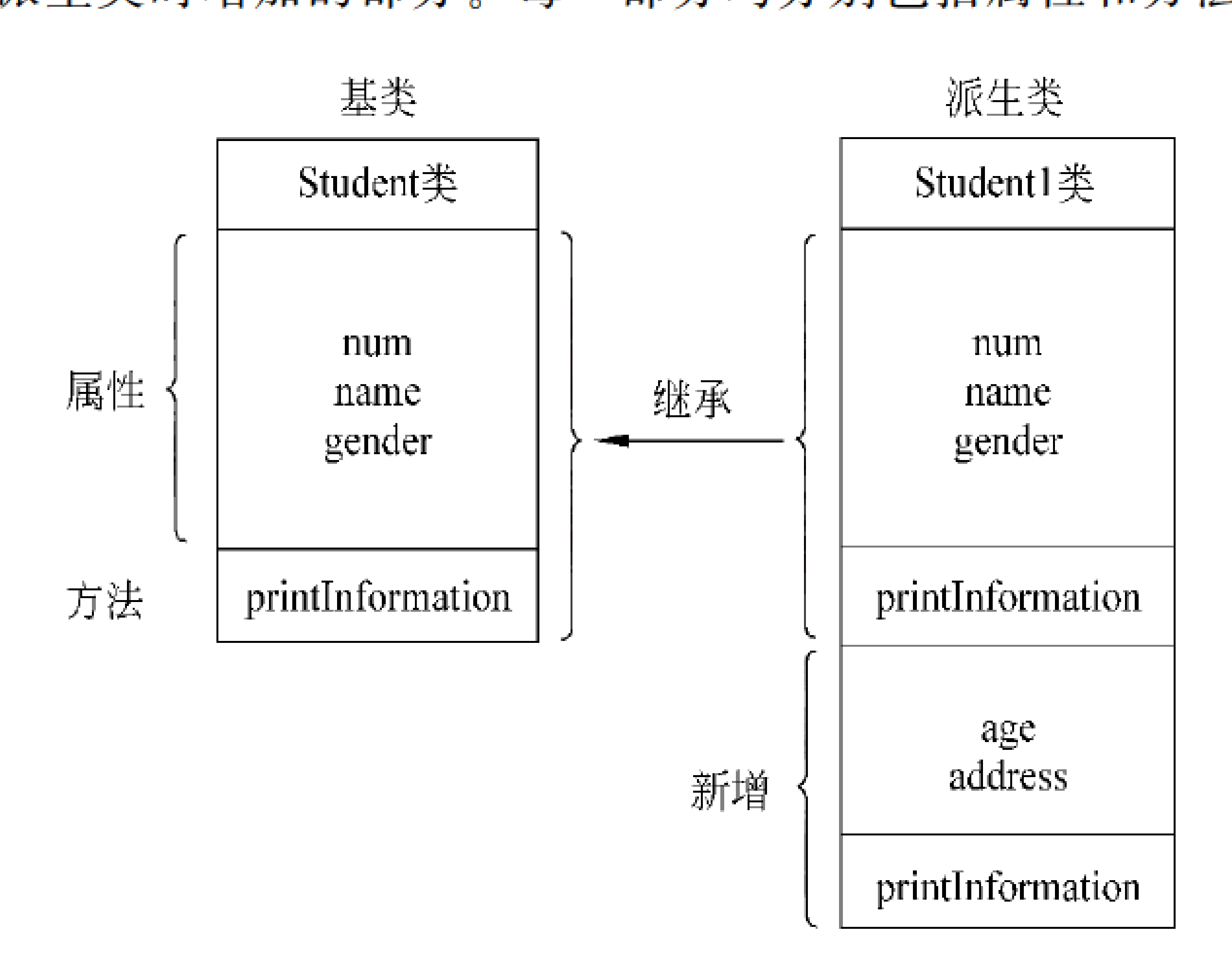


图 7-8　派生类与基类的属性和方法的继承关系

实际上，并不是把基类的属性和方法与派生类自己增加的属性和方法简单地加在一起就成为派生类。构造一个派生类包括以下三部分工作。

(1) 从基类接收属性和方法。派生类把基类全部的属性和方法接收过来，也就是说

没有选择地,不能选择接收其中一部分属性和方法,而舍弃另一部分属性和方法。

这样就可能出现一种情况:有些基类的属性和方法,在派生类中是用不到的,但是也必须继承过来。这就会造成数据的冗余,尤其是在多次派生之后,会在许多派生类对象中存在大量无用的数据,不仅浪费了大量的空间,而且在对象的建立、赋值、复制和参数的传递中,花费了许多无谓的时间,从而降低效率。这就要求我们根据派生类的需要慎重选择基类,使冗余量最小。不要随意地从已有的类中选择一个作为基类去构造派生类,应当考虑怎样能使派生类有更合理的结构。事实上,有些类是专门作为基类而设计的,在设计时充分考虑到派生类的要求。

(2) 调整从基类接收的属性和方法。接收基类的属性和方法是程序人员不能选择的,但是程序人员可以对这些属性和方法做某些调整。例如可以在派生类中声明一个与基类同名的属性、方法,则派生类中的新属性、新方法会覆盖基类的同名属性、方法。但应注意:如果是方法,不仅应使方法名相同,而且方法的参数表也相同,如果不相同,就成为方法的重载而不是覆盖了。通过这种方式可以用新属性、新方法取代基类中的属性、方法。

(3) 在定义派生类时增加的属性和方法。这部分内容是很重要的,它体现了派生类对基类功能的扩展。要根据需要仔细考虑应当增加哪些属性和方法,精心设计。例如例7-9 的例子,基类的 printInformation 方法的作用是输出学号、姓名和性别,在派生类中要求输出学号、姓名、性别、年龄和地址,不必单独写一个输出这 5 个数据的方法,而要利用基类的 printInformation 方法输出学号、姓名和性别,另外再定义一个 printInformation1 方法输出年龄和地址,先后执行这两个方法。也可以在 printInformation1 方法中调用基类的 printInformation 方法,再输出另外两个数据,在主函数中只需调用一个 printInformation1 方法即可,这样可能更清晰一些,易读性更好。例 7-9 就是采用这种方式输出这 5 个数据的。

通过以上的介绍,可以看到:派生类是基类定义的延续。可以先定义一个基类,在此基类中只提供某些最基本的功能,而另外有些功能并未实现,然后在定义派生类时加入某些具体的功能,形成适用于某一特定应用的派生类。通过对基类定义的延续,将一个抽象的基类转化成具体的派生类。因此,派生类是抽象基类的具体实现。

7.5.4 多重继承

前面讨论的是单继承,即一个类从一个基类派生而来的。实际上,常常有这样的情况:一个派生类有两个或多个基类,派生类从两个或多个基类中继承所需的属性和方法。例如,不少学校的领导干部同时又是教师,他们既有干部的属性(职务、党政部门),又有教师的属性(职称、专业、授课名称)。又如,有些学生同时是青年团的干部,同时兼有学生和青年团干部的属性。Python 为了适应这种情况,允许一个派生类同时继承多个基类。这种行为称为多重继承(multiple inheritance)。

下面通过一个例子来说明多重继承。

```
coding:utf-8
#例 7-10  多重继承
```

```
class Cadre(object):
    '定义了一个 Cadre 类 (干部类)'
    def __init__(self,position,department):
        self.position=position
        self.department=department

    def printCadreInformation(self):
        print '职务:%s,部门:%s' % (self.position,self.department)

class Teacher(object):
    '定义了一个 Teacher 类 (教师类)'
    def __init__(self,title,major,subject):
        self.title=title
        self.major=major
        self.subject=subject

    def printTeacherInformation(self):
        print '头衔:%s,专业:%s,授课名称:%s' % (self.title,self.major,self.
        subject)


class CadreTeacher(Cadre,Teacher):
    '定义了一个 CadreTeacher 类 (干部教师类)'
    def __init__(self,name,gender,age,position,department,title,major,subject):
        #通过基类名调用基类的__init__方法
        Cadre.__init__(self,position,department)
        Teacher.__init__(self,title,major,subject)
        self.name=name
        self.gender=gender
        self.age=age

    def printCadreTeacherInformation(self):
        print '姓名:%s,性别:%s,年龄:%d' % (self.name,self.gender,self.age)
        #通过 super 内建函数调用基类的方法
        super(CadreTeacher,self).printCadreInformation()
        super(CadreTeacher,self).printTeacherInformation()

#创建 CadreTeacher 类的实例化对象
ct=CadreTeacher('小张','男',30,'教务处主任','教务处','教授','计算机科学与技术','
Python 程序设计')
ct.printCadreTeacherInformation()
```

程序的运行结果如下：

```
姓名:小张,性别:男,年龄:30
```

```
职务:教务处主任,部门:教务处
头衔:教授,专业:计算机科学与技术,授课名称:Python程序设计
```

该程序首先定义了两个类 Cadre 和 Teacher,Cadre 类有两个属性 position 和 department,一个__init__构造方法,用于初始化这两个属性。还有一个 printCadreInformation 方法,用于输出相应属性。Teacher 类有三个属性 title、major 和 subject,一个__init__构造方法,用于初始化这三个属性。还有一个 printTeacherInformation 方法,用于输出相应属性。然后再定义一个 CadreTeacher 类,这个类继承 Cadre 和 Teacher 类。这样,CadreTeacher 类就拥有了这两个类的所有属性和方法。在 CadreTeacher 类中又定义了自己的属性 name、gender 和 age 以及__init__构造方法和 printCadreTeacherInformation 方法。注意:在__init__构造方法中使用基类名这种非绑定的方式调用基类的方法,而不能用 super 内建函数的方式调用基类的方法。在 printCadreTeacherInformation 方法则可以用 super 内建函数的方式调用基类的方法。这就是在 7.5.2 节中提到的问题:当派生类继承两个或多个基类时,且这两个或多个基类有同名的方法,通过 super 内建函数并不能智能地分辨出调用哪个基类的方法,如果它们的参数个数相同,这会导致某些基类的方法被调用多次,而某些基类的方法一次都没被调用。如果不同,程序会抛 TypeError 异常,提示所需参数和给定参数不一致。

在说明这个问题之前,先介绍多重继承中的方法解析顺序 MRO(Method Resolution Order)。

在 Python 2.2 以前的版本,算法非常简单:采用深度优先搜索算法,从左到右进行搜索。但由于在 Python 2.2 版本中类、类型和内建类型的子类都经过全新的改造,有新的结构,这种算法不再适用,以致出现了新的 MRO 算法。但后来也发现这种算法并不完善,于是在 Python 2.3 版本中又采用了新的 MRO 算法,这种算法是依据广度优先搜索算法设计的,应用于新式类。还需要说明的是,在 Python 2.2 及以后的版本中,经典类还是采用深度优先搜索算法,从左到右进行搜索。下面举例说明这种差异。

```
coding:utf-8
#例 7-11  经典类和新式类采用的 MRO 算法
class P1(object):                                    #基类 1
    def f(self):
        print 'P1的 f方法被调用'

class P2(object):                                    #基类 2
    def f(self):
        print 'P2的 f方法被调用'

    def g(self):
        print 'P2的 g方法被调用'

class C1(P1,P2):                                     #一级派生类,从 P1、P2 派生
    def h(self):
```

```
        print 'C1 的 h 方法被调用'

class C2(P1,P2):                    #一级派生类,从 P1、P2 派生
    def g(self):
        print 'C2 的 g 方法被调用'

class GC(C1,C2):                    #二级派生类(相对于 P1,P2),从 C1、C2 派生
    pass

gc=GC()
gc.f()
gc.g()
gc.h()
```

程序运行结果如下：

```
P1 的 f 方法被调用
C2 的 g 方法被调用
C1 的 h 方法被调用
```

该程序中类的继承关系如图 7-9 所示。P1 中定义了 f 方法,P2 中定义了 f 和 g 方法,C1 定义了 h 方法,C2 定义了 g 方法。由于 P1 和 P2 都是新式类(又继承另外的类,这里是 object 类),所以其方法解析顺序采用广度优先搜索算法,从左到右进行搜索。在主函数首先创建 GC 类的实例化对象 gc,当执行 gc.f()语句时,它首先在 GC 类中查询 f 方法,没有找到,于是按从左到右的顺序找它的基类 C1,没找到,然后找基类 C2,还是没找到,接着继续沿着继承树查找 C1 的基类 P1,此时找到 f 方法,所以其搜索顺序为 GC→C1→C2→P1。输出为“P1 的 f 方法被调用”。类似的,当执行 gc.g()语句时,其搜索顺序为 GC→C1→C2。所以输出为“C2 的 g 方法被调用”。当执行 gc.h()语句时,其搜索顺序为 GC→C1。所以输出为“C1 的 h 方法被调用”。

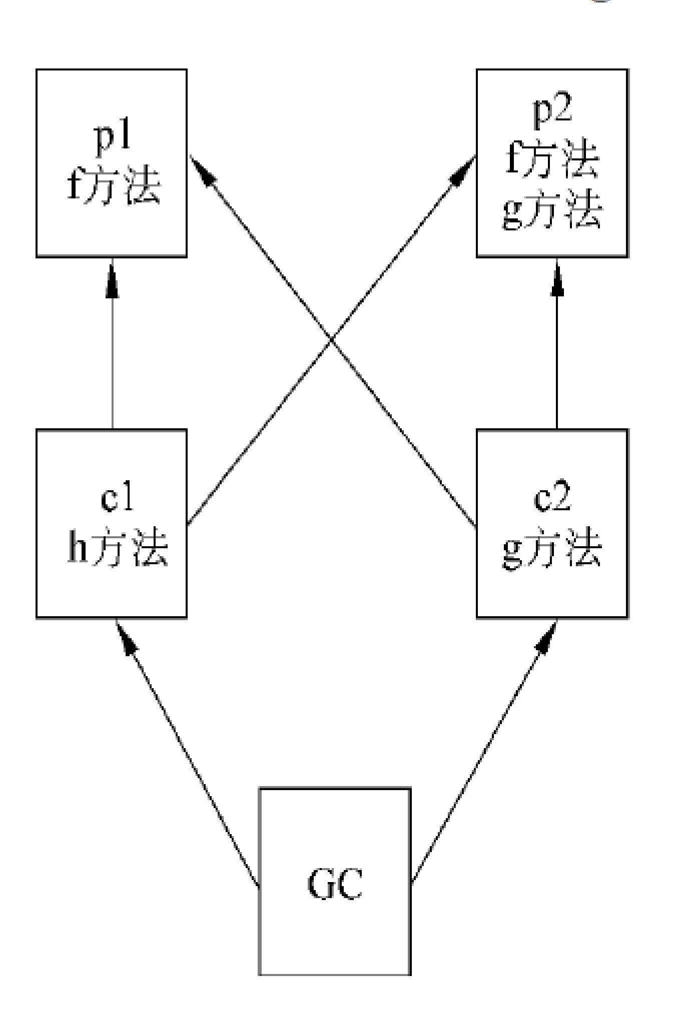


图 7-9　例 7-11 中类的继承关系

现在把 P1 和 P2 都改为经典类,即去掉它们所继承的 object 类。此时,其方法解析顺序采用深度优先搜索算法,从左到右进行搜索。程序的运行结果如下：

```
P1 的 f 方法被调用
P2 的 g 方法被调用
C1 的 h 方法被调用
```

这种情况下,当执行 gc.f()语句时,同样,它首先在 GC 类中查询 f 方法,没有找到,于是按从左到右的顺序找它的基类 C1,没找到,这时,它并不会搜索第 2 个基类 C2,而是沿着继承树查找 C1 的基类 P1,此时找到 f 方法,所以其搜索顺序为 GC→C1→P1。输出

为“P1 的 f 方法被调用”。类似的，当执行 gc.g()语句时，在 GC 类找不到 g 方法，然后在它的基类 C1 中查找，没找到，接着到 C1 的第 1 个基类 P1 中查找，还是没找到，此时 P1 已没有基类了，所以返回 C1，到 C1 的第 2 个基类 P2 中查找，在 P2 中找到了 g 函数。所以其搜索顺序为 GC→C1→P1→P2，输出为“P2 的 g 方法被调用”。当执行 gc.h()语句时，其搜索顺序为 GC→C1。所以输出为“C1 的 h 方法被调用”。

如果对深度和广度优先搜索算法不熟悉，可以查阅相关资料。对于新式类，有一个__mro__属性，这个属性是一个搜索顺序的类组成的元组，如对于上面这个例子，在新式类的情况下，执行 print GC.__mro__语句，输出结果如下：

```
(<class '__main__.GC'>,<class '__main__.C1'>,<class '__main__.C2'>,<class '__main__.P1'>,<class '__main__.P2'>,<type 'object'>)
```

有了上面关于 MRO 的知识，就可以说明 7.5.2 节中提到的问题了。但为了更形象地说明这个问题，我们结合例子进行说明。

下面分两种情况举例说明：

(1) 基类的同名方法参数个数相同的情况：

```
coding:utf-8
#例 7-12  基类的同名方法参数个数相同
class A(object):
    def __init__(self,a):
        print 'A的__init__方法被调用'
        self.a=a
        print 'A.a:%d' %self.a

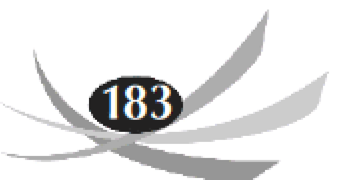
class B(object):
    def __init__(self,b):
        print 'B的__init__方法被调用'
        self.b=b
        print 'B.b:%d' %self.b

class C(A,B):
    def __init__(self,a,b):
        #调用 A的__init__方法
        super(C,self).__init__(a)
        #试图调用 B的__init__方法
        super(C,self).__init__(b)
        print 'C的__init__方法被调用'

c=C(1,2)
```

程序运行结果如下：

```
A的__init__方法被调用
A.a:1
```




```
A的__init__方法被调用
A.a:2
C的__init__方法被调用
```

可以看到基类A的__init __方法被调用了两次,而B的__init __方法一次都没被调用。这是因为它首先搜索第1个基类A,在A中找到__init __方法,停止搜索,执行A类中的__init 方法。两次调用__init __方法,两次执行的都是A类中的__init 方法,只是参数有所不同,而B的__init __方法始终未被调用。

(2) 基类的同名方法参数个数不同的情况:

```
coding:utf-8
#例7-13  基类的同名方法参数个数不同
class A(object):
    def __init__(self):
        print 'A的__init__方法被调用'
        self.a=a
        print 'A.a:%d' %self.a

class B(object):
    def __init__(self):
        print 'B的__init__方法被调用'

class C(A,B):
    def __init__(self,a):
        #调用A的__init__方法
        super(C,self).__init__(a)
        #试图调用B的__init__方法
        super(C,self).__init__()
        print 'C的__init__方法被调用'

c=C(1,2)
```

程序运行结果如下:

```
A的__init__方法被调用
A.a:1

Traceback (most recent call last):
  File "C:/Python27/7.13.py", line 19, in<module>
    c=C(1)
  File "C:/Python27/7.13.py", line 16, in __init__
    super(C,self).__init__()
TypeError: __init__() takes exactly 2 arguments (1 given)
```

显然,两次调用都找到了A类的__init__方法,第一次调用,给出一个参数,可以正常执行。第二次调用,没有给出参数,本来试图调用B类的__init__方法,结果还是调用了A

类的__init__方法,所以抛 TypeError 异常,提示参数不一致。

7.6 新式类的高级特性

通常情况下,从 object 或者其他内置类型直接或间接派生出来的类,都被称为新式类。而那些不是从 object 或者其他内置类型派生出来的类就会被当作经典类。可以看到,前面定义的绝大多数的类都是新式类。新式类是从内置类型派生出来的类,如果没有合适的内置类型可以使用,通常的做法是把 object 类作为这些新式类的基类。

相对于经典类,新式类多了一些高级特性,下面介绍其中一些比较常用的特性。

7.6.1 __slots__类属性

Python 是一门动态语言,可以在运行过程中修改对象的属性和增删方法。任何类的实例化对象都包含一个字典属性__dict__,Python 通过这个字典将任意属性绑定到对象上。但字典比较占内存资源,如果一个类有很少的属性,而有很多的实例化对象。从内存的角度上考虑,可以使用__slots__属性来代替__dict__属性。

__slots__是一个类变量(属性),是由所有合法的标识符表示的实例化对象属性组成的一个容器对象,这个容器对象可以是一个列表、元组或可迭代的对象。在设置了__slots__类属性之后,任何试图创建一个其名不在__slots__中的实例化对象属性的操作都将引发 AttributeError 异常。

下面通过一个例子来说明__slots__类属性。

```
coding:utf-8
#例 7-14  __slots__类属性
class SlotsAttribute(object):
    __slots__='name','age','gender'
    def __init__(self,name):
        self.name=name

sa=SlotsAttribute('小陈')
print '在__init__方法中初始化的属性 name:',sa.name
sa.age=20
print '在主函数创建的属性 age:',sa.age
```

程序运行结果如下:

```
在__init__方法中初始化的属性 name: 小陈
在主函数创建的属性 age: 20

Traceback (most recent call last):
  File "C:/Python27/7.14.py", line 11, in<module>
    sa.num=123456
AttributeError: 'SlotsAttribute' object has no attribute 'num'
```

这个程序的__slots__属性设置了三个标识符 name，age 和 gender。在__init__方法中可以正常创建并初始化 name 属性，在主函数中也可以正常创建 age 属性。程序的倒数第二条语句试图为实例化对象 sa 创建 num 属性，结果抛 AttributeError 异常，提示 SlotsAttribute 对象没有 num 属性，显然这是由于在__slots__类属性中并没有 nun 标识符。如果把 num 也加入到__slots__类属性中，程序能正常运行，其运行结果如下：

```
在__init__方法中初始化的属性 name: 小陈
在主函数创建的属性 age: 20
在主函数创建的属性 num: 123456
```

注意：__slots__类属性只是列出了可以创建的合法属性，但并没有创建其中的属性，需要自己根据需求手动创建。

__slots__类属性的这种特性的主要目的是节约内存，此外，它还可以防止用户随心所欲地动态增加实例化对象属性。

7.6.2 __getattribute__()特殊方法

在 Python 类中有一个特殊的方法__getattr__()，它仅当属性不能在实例化对象的__dict__、类的__dict__和其基类的__dict__中找到时才被调用。但大多数的情况是，我们想要一个方法，当访问每个属性时，该函数都被调用，而不仅仅是属性找不到才被调用的情况，这个方法就是在新式类中提供的__getattribute__()方法。

下面通过一个例子来说明__getattribute__()方法。

```
coding:utf-8
#例 7-15  __getattribute__()方法
class GetAttributeFunc(object):
    def __init__(self):
        self.default='GAF'
        self.num=123456
        self.name='小陈'
        self.age=20

    def __getattribute__(self,name):
        if name=='attr':
            print '__getattribute__方法被调用,所访问的属性为 attr'
            return self.attr
        else:
            print '访问的属性为%s,__getattribute__方法被调用' %name
            return object.__getattribute__(self,name)

    def __getattr__(self,name):
        print '访问的属性为%s,__getattr__方法被调用' %name
        return self.default
```

```
gaf=GetAttributeFunc()
gaf.num
gaf.gender
```

在上述代码中，在类 GetAttributeFunc 中使用了__init__构造方法来初始化 default、num、name 和 age 属性。然后定义__getattribute__方法，输出相关信息以确定是不是访问每个属性都调用该方法，此外，还定义了__getattr__方法，同样输出相关信息以确定该方法在什么时候被调用。最后创建 GetAttributeFunc 的实例化对象，分别访问 num 和 gender 属性，程序运行结果如下：

```
访问的属性为 num,__getattribute__方法被调用
访问的属性为 gender,__getattribute__方法被调用
访问的属性为 gender,__getattr__方法被调用
访问的属性为 default,__getattribute__方法被调用
```

从上述结果来看，当实例化对象 gaf 访问 num 属性时，__getattribute__方法被调用，访问的属性不是 attr，执行 else 子句。由于 num 属性存在实例化对象 gaf 的__dict__中，因此__getattr__方法没有被调用。当实例化对象 gaf 访问 gender 属性时，__getattribute__方法又一次被调用，同样执行 else 子句。由于 gender 属性在实例化对象 gaf、类和其基类(object)的__dict__中都找不到，因此__getattr__方法被调用，在该方法中，return 时返回 self. default，即访问了 default 属性，所以__getattribute__方法再一次被调用。

程序始终没访问 attr 属性，当访问 attr 属性时，程序会输出什么样的结果呢？显然，这会陷入一个死循环当中，可以通过调用祖先类的同名方法来避免出现这样的死循环，如__getattribute__方法中 else 子句的 object. __getattribute__(self, name)。但是这种方式只在新式类中有效。

7.6.3 描述符

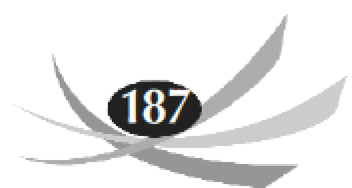
描述符是 Python 新式类中的关键符号之一。它为对象属性提供强大的 API，其原理就是将某种特殊类型的类的实例化对象指派给另一个类的属性。这个特殊的描述符类是一种新式类，包含的方法有__get__()、__set__()和__delete__()，其中__get__()方法用于得到一个属性的值，该方法在获取被指定为描述符的属性时调用；__set__()方法用于属性的赋值，该方法在为被指定为描述符的属性赋值时调用；__delete__()方法用于删除一个属性，该方法在删除被指定为描述符的属性时调用。

接下来先看一下__get__()、__set__()和__delete__()方法的原型。

```
def __get__(self,obj,typ=None)
def __set__(self,obj,val)
def __delete__(self,obj)
```

下面通过一个例子来说明新式类中的描述符：

```
coding:utf-8
#例 7-16  新式类中的描述符
```




```
class DescriptionClass(object):
    def __get__(self,obj,typ=None):
        print '__get__方法被调用,obj:%s,typ:%s' % (obj,typ)
        return self.data

    def __set__(self,obj,val):
        print '__set__方法被调用,obj:%s,val:%s' % (obj,val)
        self.data=val

    def __delete__(self,obj):
        print '__delete__方法被调用,obj:%s' %obj
        del obj

class TestDescriptionClass(object):
    this=DescriptionClass()
    that=DescriptionClass()
    other=1

tdc=TestDescriptionClass()
tdc.this='test'
print tdc.this
print '类属性 other 为',tdc.other
tdc.other=2
print '实例化对象 tdc 的属性 other 为',tdc.other
print '类属性 other 为',tdc.__class__.other
del tdc.other
print '使用 del 语句删除 that 描述符'
del tdc.that
```

程序运行结果如下:

```
__set__方法被调用,obj:<__main__.TestDescriptionClass object at 0x0000000002F349E8>,
val:test
__get__方法被调用,obj:<__main__.TestDescriptionClass object at 0x0000000002F349E8>,
typ:<class '__main__.TestDescriptionClass'>
test
类属性 other 为 1
实例化对象 tdc 的属性 other 为 2
类属性 other 为 1
使用 del 语句删除 that 描述符
__delete__方法被调用,obj:<__main__.TestDescriptionClass object at 0x0000000002F349E8>
```

该程序首先定义了 DescriptionClass 类,类中定义了__get__()、__set__()和__delete__()方法,分别执行相应的功能。然后又定义了一个 TestDescriptionClass 类,把属性 this 和 that 定义为 DescriptionClass 类的描述符,属性 other 是普通的类属性。在主函数

中,实例化 TestDescriptionClass 类的对象 tdc。可以看到,在为 this 描述符赋值时,__set__方法被调用,在获取 this 描述符时,__get__方法被调用,接着输出类的普通属性 other,然后为 other 属性赋值(实际上是创建实例化对象 tdc 的属性 other),执行一样的语句 tdc.other,得到的结果是赋值后的 2,这是因为实例化对象的属性把类属性“屏蔽”了。之后执行 tdc.__class__.other,输出值仍为 1,验证了 other 类属性并没有被改变,只是被“屏蔽”了。最后通过 del 语句删除 other 属性和 that 属性(描述符),当删除 that 属性时,__delete__方法被调用。

7.7 本章小结

本章主要讲解了以下几个知识点。

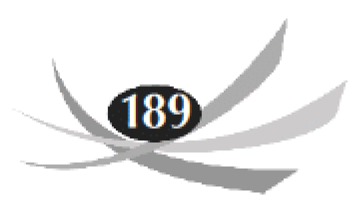
(1) 面向对象程序设计。面向对象程序设计的基本思路是将数据和对数据的操作方法集中放在一个整体中,形成一个相互依存、不可分割的整体,这个整体即为对象。通过相同类型的对象,抽象出其共性而形成类。此外,在类中必须声明一些函数(方法),这些函数用于与外界进行通信。面向对象程序设计的主要特点是封装、继承和多态。这种设计方法与传统的面向过程的方法相比,具有更好的可重用性、可扩展性和可管理性。

(2) 类和对象。类代表了某一批对象的共性和特性。类是对象的抽象,而对象是类的具体实例。通过 class 关键字可以定义类,根据其是否继承内建类型的类,可以把类分为经典类和新式类。创建类的实例化对象和调用函数类似,即类名(),如果定义的__init__构造方法有参数,还需要传入相应的参数。

(3) 类、对象的属性和方法。在类内,且在方法外定义的,无特别声明的变量称为类属性,或者称为静态属性。类属性既可以通过类名来访问,又可以通过对象名来访问。对象属性要放在方法中声明,且有对象名(通常为 self).前缀,只能通过对象名访问。无论是类属性还是对象属性,都可以根据访问的权限分为公有属性和私有属性。以两个下划线开头声明的属性称为私有属性,否则称为公有属性。访问私有属性的语法:类(对象)名._类名__私有属性名。类方法可以通过@classmethod 指令的方式定义或者通过使用内建函数 classmethod 的方式将一个普通的方法转为类方法。对象方法其实就是在类中没有通过特殊指令定义的普通方法。同样地,类方法和对象方法也可以根据访问的权限分为公有方法和私有方法。以两个下划线开头声明的方法称为私有方法,否则称为公有方法。访问私有方法的语法:类(对象)名._类名__私有方法名。此外,还有静态方法,定义静态方法时可以通过@staticmethod 指令的方式定义或者通过使用内建函数 staticmethod 的方式将一个普通的方法转为静态方法。调用时,无论是对象方法、类方法还是静态方法,都可以通过类或者对象的方式调用。

(4) 内置属性和方法。内置属性,如__doc__、__dict__和__bases__等,是由 Python 解析器提供的,用于管理类的内部关系。内置方法,如__init__()、__str__()和__getattr__()等也是由 Python 解析器提供的。这些内置方法中的一部分有默认的行为,而另一部分则没有,留到需要的时候去实现。这些内置方法是 Python 中用来扩展类的强有力的方式。

(5) 类的组合。类的组合就是让不同的类混合并入其他的类,形成更加复杂、更符合



需求的类，从而增加功能和提高代码重用性。

(6) 类的继承与派生。一个新类从已有的类那里获得其已有特性，这种现象称为类的继承。从另一个角度来说，从已有的类(父类)产生一个新的子类，称为类的派生。类的继承是用已有的类来建立专用类的编程技术。派生类继承了基类的所有属性和方法，并可以对属性和方法做必要的增加和调整。

(7) 派生类的组成。一个派生类包括从基类接收的属性和方法、调整从基类接收的属性和方法以及在定义派生类时增加的属性和方法。

(8) 多重继承。多重继承就是一个类继承自两个或多个基类，从这些基类中继承所需的属性和方法。多重继承中的方法解析顺序 MRO 在经典类和新式类中有不同的表现。在经典类中采用深度优先搜索算法，从左到右进行搜索，而在新式类中则采用不同的搜索算法，这种算法是依据广度优先搜索算法设计的。

(9) 新式类的高级特性。新式类中的__slots__类属性是由所有合法的标识符表示的实例化对象属性组成的一个容器对象。在设置了__slots__类属性之后，任何试图创建一个其名不在__slots__中的实例化对象属性的操作都将引发 AttributeError 异常。__slots__类属性的这种特性的主要目的是节约内存，此外，它还可以防止用户随心所欲地动态增加实例化对象属性。新式类中的__getattribute__()方法，当访问每个属性时，该方法都被调用，而不像__getattr__()方法那样仅仅是属性找不到才被调用。新式类中的描述符为对象属性提供强大的 API。这个特殊的描述符类是一种新式类，包含的方法有__get__()、__set__()和__delete__()，其中__get__()方法用于得到一个属性的值，该方法在获取被指定为描述符的属性时调用；__set__()方法用于属性的赋值，该方法在为被指定为描述符的属性赋值时调用；__delete__()方法用于删除一个属性，该方法在删除被指定为描述符的属性时调用。

7.8　习　　题

一、解答题

1. 什么是面向对象程序设计？面向对象的基本思路是什么？面向对象程序设计的特点和优势又是什么？

2. 什么是类？什么是对象？类和对象又有什么关系？

3. 类属性和对象属性是一个概念吗？属性根据访问权限可以分为什么？类中定义的方法又可以分为什么？

4. 有哪些类的内置属性和方法？它们的作用分别是什么？

5. 什么是继承？继承的优势是什么？

6. 派生类由哪三部分组成？

7. 多重继承中的方法解析顺序 MRO 在经典类和新式类中有什么不同的表现？

8. 新式类有哪些主要的高级特性？

二、看程序写结果

1.

```
class Person(object):
    '定义了一个 Person 类'
    __nation='中国'
    city='厦门'
    def __init__(self,name,age,gender):
        self.name=name
        self.age=age
        self.__gender=gender

p=Person('小陈',20,'男')
print 'nation:%s,city:%s' % (Person._Person__nation,Person.city)
print 'nation:%s,city:%s' % (p._Person__nation,p.city)
print 'name:%s,age:%d,gender:%s' % (p.name,p.age,p._Person__gender)
p.city='北京'
print 'nation:%s,city:%s' % (Person._Person__nation,Person.city)
print 'nation:%s,city:%s' % (p._Person__nation,p.city)
print 'nation:%s,city:%s' % (p._Person__nation,p.__class__.city)
```

2.

```
class Methods(object):
    '定义一个 Methods 类'
    @classmethod
    def publicClassMethod(cls):
        print 'called publicClassMethod'
        return cls

    @staticmethod
    def __privateStaticMethod():
        return 'called privateStaticMethod!'

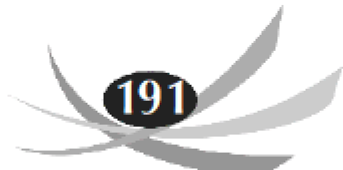
    def publicMethod(self):
        print 'called publicMethod!'

    def __privateMethod(self):
        print 'called privateMethod!'

    privateMethodToClassMethod=classmethod(__privateMethod)

    publicMethodToStaticMethod=staticmethod(publicMethod)


m=Methods()
Methods.publicClassMethod()
m.publicClassMethod()
```




```
Methods._Methods__privateStaticMethod()
m._Methods__privateStaticMethod()

Methods.publicMethod(m)
m.publicMethod()

Methods._Methods__privateMethod(m)
m._Methods__privateMethod()

Methods.privateMethodToClassMethod()
m.privateMethodToClassMethod()

Methods.publicMethodToStaticMethod(m)
m.publicMethodToStaticMethod(m)
```

3.

```
class P1(object):
    def f(self):
        print 'P1的 f方法被调用'

class P2(object):
    def f(self):
        print 'P2的 f方法被调用'

    def g(self):
        print 'P2的 g方法被调用'

class C1(P2,P1):
    def h(self):
        print 'C1的 h方法被调用'

class C2(P2,P1):

    def g(self):
        print 'C2的 g方法被调用'

class GC(C2,C1):
    pass

gc=GC()
gc.f()
gc.g()
gc.h()
```

(把 P1,P2 改为经典类,结果又如何?)

4.

```
class NewStyleClass(object):
    __slots__='name','age','gender','num','data'
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def __get__(self,obj,typ=None):
        print '__get__方法被调用'
        return self.data

    def __set__(self,obj,val):
        print '__set__方法被调用'
        self.data=val

    def __delete__(self,obj):
        print '__delete__方法被调用'
        del obj

class TestClass(object):
    this=NewStyleClass('小张',21)
    other=12

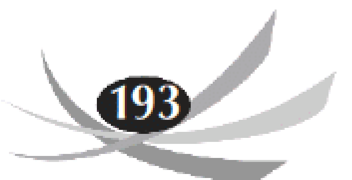
tc=TestClass()
tc.this=NewStyleClass('小李',25)
print 'name:%s,age:%d' % (tc.this.name,tc.this.age)
print tc.other
tc.other=21
print tc.__class__.other
print tc.other
del tc.other
print 'del'
del tc.this
```

三、上机练习

1. 设计一个类 Student,在类中定义两个方法,一个方法用于输入某个学生的三门成绩,另一个方法计算该学生的总分和平均分,并输出。

2. 设计一个类 Bank,实现银行某账号的资金往来账目管理,包括建账号、存入和取出,分别通过三个方法实现。

3. 设计一个类 Attribute,类中有公有类属性(pubClaAttr),私有类属性(priClaAttr),并有初始化,在__init__构造方法中初始化公有对象属性(pubObjAttr)和私有对象属性(priObjAttr),在主函数中创建类 Attribute 的实例化对象,然后通过类名和对象名的方式访问类属性,通过对象名访问对象属性。



4. 设计一个类 Methods，类中定义有公有类方法（pubClaMet）、私有类方法（priClaMet）、公有对象方法（pubObjMet）、私有对象方法（priObjMet）、公有静态方法（pubStaMet）和私有静态方法（priStaMet）。在主函数中创建类 Methods 的实例化对象，然后通过类名和对象名的方式访问这些方法。

5. 设计一个电脑类 Computer，类中的属性有主机类和显示器类。其中，主机类的属性又有主板类，内存类和硬盘类。显示器类有价格和分辨率属性，主板类有价格和型号属性，内存类有价格和容量属性，硬盘类也有价格和容量属性。在显示器类、主板类、内存类和硬盘类中都有 printMessage 方法，分别输出它们相应的属性。在主机类也有 printMessage 方法，在该方法中调用主板类、内存类和硬盘类中的 printMes sage 方法，在电脑类中的 printMessage 方法调用主机类和显示器类的 printMess age 方法。在主函数创建 Computer 的实例化对象，然后调用 printMessage 方法输出所有的电脑信息。

6. 首先定义一个 Point（点）类，包含属性 x，y（坐标点），方法有 setPoint、getX、getY、calDistance 和 printPointInfo，它们的作用分别为设置坐标值，获取 x 坐标，获取 y 坐标，计算点的距离和输出点的信息（包括坐标和距离）。以它为基类，派生出一个 Circle（圆）类，增加属性 r（半径），该类的方法有 setRadius、getRadius、calArea 和 printCircleInfo，它们的作用分别为设置圆的半径，获取圆的半径，计算圆的面积和输出圆的信息（包括原点、半径和面积）。再以 Circle 类为直接基类，派生出一个 Cylinder（圆柱体）类，再增加属性 h（高），该类的方法有 setHeight、getHeight、calArea、calVolume 和 printCylinderInfo，它们的作用分别为设置圆柱的高、获取圆柱的高、计算圆柱的表面积、计算圆柱的体积和输出圆柱的信息（包括原点、半径、表面积和体积）。

7. 分别定义 Teacher（教师）类和 Cadre（干部）类，采用多重继承方式由这两个类派生出新类 TeacherCadre（教师兼干部）。在两个基类中都有姓名、年龄、性别、地址、电话属性，且这些属性都使用相同的名字。在 Teacher 类中还有 title（职称）属性和 printTeacherInfo 方法，该方法用于输出 Teacher 类中的属性信息。在 Cadre 类中还有 position（职务）属性和 printCadreInfo 方法，该方法用于输出 Cadre 类中的属性信息。在 TeacherCadre 类中还有 wages（工资）属性。此外，还有 printTeacherCadreInfoByT 方法和 printTeacherCadreInfoByC 方法。在 printTeacherCadreInfoByT 方法中调用 Teacher 类的 printTeacherI nfo 方法，输出姓名、年龄、性别、职称、地址和电话，然后再用 print 语句输出职务和工资。在 printTeacherCadreInfoByC 方法中调用 Cadre 类的 printCadreI nfo 方法，输出姓名、年龄、性别、职务、地址和电话，然后再用 print 语句输出职称和工资。

第 8 章

模 块 和 包

本章学习目标

- 理解命名空间的相关概念
- 掌握模块及模块的导入
- 了解模块导入的特性及模块内建函数
- 掌握包的相关概念

在第 6 章中已经提到，Python 程序是由包、模块、函数组成的。其中，包是由一系列模块组成的集合，而模块是处理某一类问题的函数或(和)类的集合。函数和类已在前面的章节中介绍过。本章首先介绍命名空间的相关概念，然后再集中介绍 Python 模块、包以及如何把模块和包导入到当前的编程环境中，同时也会涉及与模块、包相关的概念。

8.1 命 名 空 间

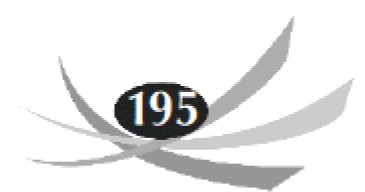
命名空间是从名称(变量或者说是标识符)到对象的映射。当一个名称映射到一个对象上时，这个名称和这个对象就绑定了。我们可以把命名空间理解为一个容器，在这个容器中可以装许多名称。

8.1.1 命名空间的分类

Python 中一切都是对象，如整数、字符串、列表等数据，此外，还包括函数、模块、类和包本身，这些对象都存在于内存中。但是我们怎么找到所需的对象呢？这就需要首先判断所找的对象所处的命名空间。Python 中有三类命名空间：内建命名空间(builtin namespace)、全局命名空间(global namespace)和局部命名空间(local namespace)。在不同的命名空间中的名称是没有关联的。此外，不同的全局命名空间或不同的局部命名空间，它们的名称也是没有关联的。

每个对象都有自己的命名空间，可以通过对象.名称的方式访问对象所处的命名空间下的名称，每个对象的名称都是独立的。即使不同的命名空间中有相同的名称，它们也是没有任何关联的。

命名空间都是动态创建的，并且每一个命名空间的生存时间也不一样。内建命名空间是在 Python 解释器启动时创建，一直存在当前编程环境中，直到退出解析器。全局命名空间在读入模块定义时，即该模块被导入(import)的时候创建，通常情况下，全局命名



空间也会一直保存到解释器退出。局部命名空间在函数(或类的方法)被调用时创建,在函数返回或者引发了一个函数内部没有处理的异常时删除。

8.1.2　命名空间的规则

理解 Python 的命名空间需要掌握以下三条规则:

第一,赋值语句(包括显式赋值和隐式赋值)会把名称绑定到指定对象中,赋值的地方决定名称所处的命名空间。

第二,函数、类定义会创建新的命名空间。

第三,Python 搜索一个名称的顺序是“LEGB”。

所谓的“LEGB”是 Python 中四层命名空间的英文名字首字母的缩写,这里的四层命名空间是上面所说的三个命名空间的一个细分。

第一层是 L(local),表示在一个函数定义中,而且在这个函数里面没有再包含函数的定义。

第二层是 E(enclosing function),表示在一个函数定义中,但这个函数里面还包含有函数的定义,其实 L 层和 E 层只是相对的,这两层空间合起来就是上面所说的局部命名空间。

第三层是 G(global),表示一个模块的命名空间,也就是说在一个. py 文件中,且在函数或类外构成的一个空间,这一层空间对应上面所说的全局命名空间。

第四层是 B(builtin),表示 Python 解释器启动时就已经加载到当前编程环境中的命名空间,之所以叫 builtin 是因为在 Python 解释器启动时会自动载入__builtin__模块,这个模块中的 list、str 等内置函数就处于 B 层的命名空间中,这一层空间对应上面所说的内建命名空间。

8.1.3　命名空间的例子

下面通过一个例子来理解命名空间。

```
coding:utf-8
#例 8-1　命名空间
a=int('12')
def outFunc():
    print '调用 outFunc 函数'
    b=3
    a=4
    def inFunc():
        print '调用 inFunc 函数'
        b=5
        c=a+b
        print '调用 inFunc 函数的返回值为',c
        return c
    e=b+inFunc()
```

```
        print '调用 outFunc 函数的返回值为',e
        return e

outFunc()
```

程序运行结果如下:

```
调用 outFunc 函数
调用 inFunc 函数
调用 inFunc 函数的返回值为 9
调用 outFunc 函数的返回值为 10
```

首先我们看一下该程序中各命名空间是什么时候被创建的。把该程序保存成一个.py文件,然后启动 Python 解析器,此时内建命名空间和全局命名空间被创建,在主函数调用 outFunc 函数时创建局部命名空间。

接下来我们分析该程序中的各个名称处于什么命名空间:

第 3 行,赋值语句,适用第一条规则,把名称 a 绑定到由内建函数 int 创建的整型 3 这个对象。赋值的地方决定名称所处的命名空间,因为 a 是在函数外定义的,因此 a 处于 G 层命名空间中,即全局命名空间。注意,这一行中还有一个名称,那就是 int。由于 int 是内置函数,是在__builtin__模块中定义的,所以 int 就处于 B 层命名空间中,即内建命名空间。

第 4 行,由于 def 中包含一个隐性的赋值过程,适用第一条规则,把名称 outFunc 绑定到所创建的函数对象中。由于 outFunc 是在函数外定义的,因此 outFunc 处于 G 层命名空间中。此外,这一行还适用第二条规则,函数定义会创建新的命名空间(局部命名空间)。

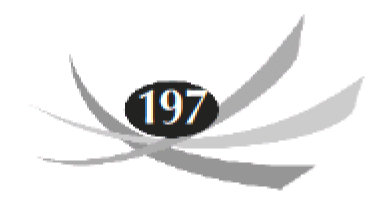
第 6 行,适用第一条规则,把名称 b 绑定到 3 这个对象中。由于这是在一个函数内定义,并且内部还有函数定义,因此 b 处于 E 层命名空间中,精确来说是处于 outFunc 函数创建的局部命名空间。

第 7 行,适用第一条规则,把名称 a 绑定到 4 这个对象中。注意,这个名称 a 与 b 名称一样都处于 E 层命名空间中。但这个名称 a 与第 3 行的名称 a 是不同的,因为它们所处的命名空间是不一样的。

第 8 行,适用第一条规则,把名称 inFunc 绑定到所创建的函数对象中。由于名称 inFunc 是在 outFunc 函数内定义的,因此名称 inFunc 处于 L 层命名空间中,即定义 outFunc 函数时创建的局部命名空间。同样,函数定义也会创建新的局部命名空间。

第 10 行,适用第一条规则,把名称 b 绑定到 5 这个对象中。由于这个名称 b 是在 inFunc 函数内定义的,而且在这个函数内部没有其他的函数定义,因此这个名称 b 处于 L 层命名空间中,精确来说是处于 inFunc 函数创建的局部命名空间中。这个名称 b 和第 6 行中的名称 b 是不同的,它们分别处于 inFunc 函数创建的局部命名空间和 outFunc 函数创建的局部命名空间(尽管它们都处于局部命名空间)。

第 11 行,适用第三条规则,Python 解释器首先识别到名称 a,按照 LEGB 的顺序查找,先找 L 层(也就是在 inFunc 内部),没有找到,再找 E 层(也就是在 outFunc 内部,inFunc 外),找到,其值为 4。然后又识别到名称 b,同样按 LEGB 的顺序查找,在 L 层找到,其值为 5,然后把 4 和 5 相加得到 9,紧接着创建 9 这个对象,把名称 c 绑定到这个对



象中。和第10行的b一样,这个名称c也处于L层命名空间。

后面的语句类似,这里就不再一一分析了。其实,所谓的“LEGB”是为了学术上便于表述而创造的。让一个编程的人说出哪个名称处于哪个层没有什么意义,只要知道对于一个名称,Python是怎么寻找它的值的就可以了。

通过上面的例子也可以看出,如果在不同的命名空间中定义了相同的名称是没有关系的,并不会产生冲突。寻找一个名称的过程总是从当前层(命名空间)开始查找,如果找到就停止查找,没有找到就往上层查找,直到找到为止,或者抛查找不到的异常。由此可以这么说,B层内的名称在所有模块(.py文件)中可用,G层内的名称在当前模块(.py文件)中可用,E和L层内的名称在当前函数内可用。

8.2 模　　块

8.2.1 什么是模块

模块支持从逻辑上组织Python代码。当代码量变得相当大的时候,我们最好把代码分成一些有组织的代码段,并保证它们之间彼此的关联性。这些代码段可能是一个包含属性和方法的类,也可能是一组相关但彼此独立的操作函数。这些代码段是共享的,所以Python允许导入一个之前已经编写好的模块,以实现代码的重用。这个把其他模块中的名称(变量)、函数和类附加到当前的模块中的操作就称为导入,而这些有组织,实现某些功能的代码段就是模块。

模块是把一组相关的名称、函数、类或者是它们的组合组织到一个文件中。如果说模块是按照逻辑来组织Python代码的方法,那么文件便是物理层上组织模块的方法。因此一个文件被看作一个独立的模块,一个模块也可以被看作一个文件。模块的文件名就是模块的名字加上扩展名.py。

8.2.2 导入模块

1. import语句

使用import语句导入模块时,其语句如下:

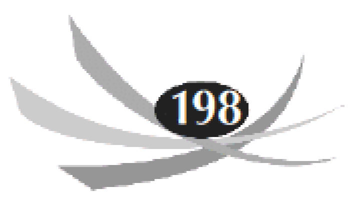
```
import module1
import module2
    ⋮
import moduleN
```

也可以在一行内导入多个模块,如:

```
import module1 [,module2 [,moduleN]]
```

但是这样的代码可读性不如多行的导入语句。而且在性能上和生成Python字节码时这两种做法没有什么不同。所以一般情况下,我们使用第一种导入格式。

我们推荐所有的模块在Python模块的开头部分导入。而且导入顺序最好按照



Python 标准库模块、Python 第三方模块、应用程序自定义模块的顺序导入，并且使用一个空行分隔这三类模块的导入语句。这将确保模块使用固定的顺序导入，有助于减少每个模块需要的 import 语句数目。

模块的导入需要一个称为“路径搜索”的过程。即在文件系统“预定义区域”中查找所要导入的模块对应的文件名。这些预定义区域其实就是 Python 搜索路径的集合。注意，路径搜索和搜索路径是两个不同的概念，前者是指查找某个文件的操作，后者是去查找一组目录。

有时候导入模块操作会失败，如：

```
>>> import mymodule
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
ImportError: No module named mymodule
```

发生这样的错误时，解析器会提示说无法访问请求的模块，可能的原因是模块不在搜索路径里，从而导致了路径搜索的失败。

默认搜索路径是在安装 Python 或者是编译程序时指定的，可以在系统的 PYTHONPATH 环境变量中修改默认搜索路径。该变量的内容是一组用冒号分隔的目录路径。如果想让解析器使用这个变量，那么要确保在启动解析器或执行 Python 脚本前设置了该变量。

当然了，解析器启动之后，也可以访问这个搜索路径，它被保存在 sys 模块的 sys. path 变量中。注意，这个变量是一个包含每个独立路径的列表。下面是本机 sys. path 的内容，注意，不同的系统，搜索路径一般都不同。

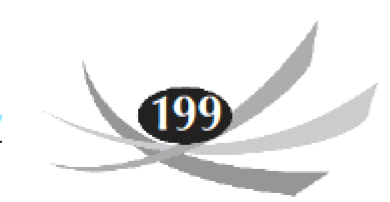
```
>>> import sys
>>> sys.path
['', 'C:\\WINDOWS\\SYSTEM32\\python27.zip', 'C:\\Python27\\DLLs', 'C:\\Python27\\lib', '
C:\\Python27\\lib\\plat-win', 'C:\\Python27\\lib\\lib-tk',
'C:\\Python27', 'C:\\Python27\\lib\\site-packages']
```

这只是一个列表，我们可以根据需要对它进行修改。如果你要导入的模块，其对应的路径不在这个搜索路径中，这时可以使用列表的 append()方法或者 insert()方法把对应的路径添加到搜索路径中，如：

```
sys.path.append('C:\\home')
```

修改完成后，就可以加载自己的模块了。只要这个列表中的某个目录包含这个文件，该模块就会被正确导入。如：

```
>>> import sys
>>> import mymodule
Traceback (most recent call last):
  File "<stdin>", line 1, in<module>
ImportError: No module named mymodule
```




```
>>> sys.path.append('C:\\home')
>>> sys.path
['', 'C:\\WINDOWS\\SYSTEM32\\python27.zip', 'C:\\Python27\\DLLs', 'C:\\Python27\\lib', '
C:\\Python27\\lib\\plat-win', 'C:\\Python27\\lib\\lib-tk',
'C:\\Python27', 'C:\\Python27\\lib\\site-packages', 'C:\\home']
>>> import mymodule
>>>
```

使用 sys. modules 可以查看当前导入了哪些模块和它们来自什么地方。sys. modules 是一个字典，使用模块名作为键(key)，对应的物理地址作为值(value)。

Python 解析器执行到 import 语句时，如果在搜索路径中找到指定的模块，就会加载它。该过程遵循作用域原则，如果在一个模块的顶级导入，那么它的作用域就是全局的；如果在函数导入，那么它的作用域就是局部的。

模块可以被导入多次，但只有第一次导入被加载并执行。

2. from-import

可以在模块里导入指定模块的属性(变量、函数或者类等)，也就是把指定变量、函数或者类导入到当前作用域中。使用 from-import 语句可以实现这个目的，其语法如下：

```
from module import name1 [,name2 [,... nameN]]
```

当导入的属性有很多时，import 行会越来越长，直到自动换行，而且需要一个反斜杠“\”。如：

```
from Tkinter import Tk, Frame, Button, Entry, Canvas, Text, LEFT \
            DISABLED, NORMAL, RIDGE, END
```

当然，我们可以选择多行的 from-import 语句，如：

```
from Tkinter import Tk, Frame, Button, Entry, Canvas, Text, LEFT
from Tkinter import DISABLED, NORMAL, RIDGE, END
```

如果需要把指定模块的所有属性都导入到当前名称空间，可以使用如下语法：

```
from Tkinter import *
```

但不建议过多地使用这种方式，因为它会“污染”当前的名称空间，而且很可能覆盖当前名称空间中现有的名字，如果某个模块有很多要经常访问的变量或者模块的名字很长，这也不失为一个方便的办法。建议只在两种场合下使用这样的方式导入：一是目标模块中的属性非常多，反复输入模块名很不方便；另一个是在交互式解析的场合下，因为这样可以减少输入的次数。

3. 扩展的 import 语句

有时候导入的模块或是模块属性名称已经在程序中使用了，又或者是因为其名称太长，我们不想使用导入的名字。Python 为了迎合这个需求，提供了扩展的 import 语句，在导入时指定局部绑定名称。如：

```
import MyModule as mm
```

```
from MyModule import StudentClass as sc
```

下面通过一个例子来理解模块导入。

实现 hanoi 塔的程序，文件名：hanoiFunction.py。

```
#实现 hanoi 塔的函数,文件名:hanoiFunction.py
#coding:utf-8
#例 8-2(1)  模块导入
def hanoi(n,one,two,three):
    if n==1:
        move(one,three)
    else:
        hanoi(n-1,one,three,two)
        move(one,three)
        hanoi(n-1,two,one,three)

def move(x,y):
    print '%c->%c' % (x,y)

def hanoiMain():
        n=input('请输入盘子的数目:')
        print '移动'+str(n)+'个盘子的步骤如下:'
        hanoi(n,'A','B','C')
```

多重继承的程序，文件名：multipleInheritance.py。

```
#多重继承的程序,文件名:multipleInheritance.py
#coding:utf-8
#例 8-2(2)  模块导入
class Cadre(object):
    '定义了一个 Cadre 类(干部类)'
    def __init__(self,position,department):
        self.position=position
        self.department=department

    def printCadreInformation(self):
        print '职务:%s,部门:%s' % (self.position,self.department)

class Teacher(object):
    '定义了一个 Teacher 类(教师类)'
    def __init__(self,title,major,subject):
        self.title=title
        self.major=major
        self.subject=subject

    def printTeacherInformation(self):
```




```
        print '头衔:%s,专业:%s,授课名称:%s' % (self.title,self.major,self.subject)


class CadreTeacher(Cadre,Teacher):
    '定义了一个 CadreTeacher 类(干部教师类)'
    def __init__(self,name,gender,age,position,department,title,major,subject):
        #通过基类名调用基类的__init__方法
        Cadre.__init__(self,position,department)
        Teacher.__init__(self,title,major,subject)
        self.name=name
        self.gender=gender
        self.age=age

    def printCadreTeacherInformation(self):
        print '姓名:%s,性别:%s,年龄:%d' % (self.name,self.gender,self.age)
        #通过 super 内建函数调用基类的方法
        super(CadreTeacher,self).printCadreInformation()
        super(CadreTeacher,self).printTeacherInformation()

def multipleInheritanceMain():
        #创建 CadreTeacher 类的实例化对象
        ct=CadreTeacher('小张','男',30,'教务处主任','教务处','教授','计算机科学与技术
        ','Python 程序设计')
        ct.printCadreTeacherInformation()
```

主模块,文件名:importModuleTest.py。

```
#主模块,文件名:importModuleTest.py
#例 8-2(3)  模块导入
import hanoiFunction
from multipleInheritance import multipleInheritanceMain as miMain
print '----------------调用 hanoiFunction 模块的主函数-----------------'
hanoiFunction.hanoiMain()
print '----------------调用 multipleInheritance 模块的主函数-----------'
miMain()
```

程序第1行通过 import 语句导入了 hanoiFunction,第4行就可以调用该模块的 hanoiMain 函数,第2行通过 from-import 语句导入 multipleInheritance 模块的 multipleInheritanceMain 函数,并改名为 miMain,第6行就可以直接调用该函数。程序运行结果如下:

```
----------------调用 hanoiFunction 模块的主函数-----------------
请输入盘子的数目:3
移动 3个盘子的步骤如下:
A→C
```

```
A→B
C→B
A→C
B→A
B→C
A→C
-----------------调用 multipleInheritance 模块的主函数-----------
姓名:小张,性别:男,年龄:30
职务:教务处主任,部门:教务处
头衔:教授,专业:计算机科学与技术,授课名称:Python 程序设计
```

8.2.3 模块导入的特性

1. 载入时执行模块

加载模块会同时执行该模块,也就是说被导入模块的顶级(主函数)代码将直接被执行,包括全局变量以及类和函数的声明等,当然,这样的结果可能不是我们想要的。为尽量避免这种情况的发生,我们应该尽可能地把代码封装到函数或类中。

2. 导入和加载

一个模块,无论它被导入多少次,都只会被加载一次,这可以防止多重导入时代码被多次执行。例如你的模块导入了 sys 模块,而你要导入的其他模块也导入了 sys 模块,尽管 sys 模块被导入多次,但只在第一次导入时加载该模块并执行。

3. __future__特性

任何一门语言都是动态发展的语言,Python 也不例外。Python 为了满足程序开发的需求,使开发过程变得更加灵活、便捷,会对之前的特性不断进行改进,完善,并提出新的特性,而其中的某些变化可能会影响到当前的功能,为使开发出来的程序有更好的扩展性和稳定性,满足未来语言发展的变化,Python 提供了__future__特性,以让程序员提前体验增强特性或新特性的变化,在特性固定下来的时候修改程序,其语法如下:

```
from __future__ import new_future
```

注意:不能通过 from __future__导入所有特性,而必须要指定导入哪个特性。

8.2.4 模块内建函数

1. __import__()

Python 1.5 加入了__import__()函数,实际上,它是作为导入模块的函数,也就是说 import 语句调用了该函数来实现模块的导入,提供这个函数是为了让有特殊需要的用户可以覆盖它,实现自定义的导入算法。

__import__()的语法如下:

```
__import__(module_name [,globals [,locals [,fromlist]]])
```

其中,module_name 是要导入的模块名称,globals 是包含当前全局符号表的名字字典,locals 是包含局部符号表的名字字典,fromlist 是一个使用 from-import 语句所导入符



号的列表。globals、locals 和 fromlist 参数都是可选的，默认值分别为 globals()、locals()和[]。

导入 sys 模块可以通过下面的语句实现：

```
sys=__import__('sys')
```

2. globals()和 locals()

globals()和 locals()内建函数分别返回调用处可以访问的全局和局部命名空间中的名称组成的字典。在一个函数内部，局部命名空间代表在函数执行的时候定义的所有名字，locals()函数返回的就是包含这些名字组成的字典。glocals()会返回函数可访问的全局名字组成的字典。

在全局命名空间下，globals()和 locals()返回相同的字典，因为这时的局部命名空间就是全局命名空间。

下面通过一个例子来理解这两个函数：

```
#coding:utf-8
#例 8-3  globals()和 locals()函数
def func():
    print '调用 func 函数'
    anum=123
    astring='abc'
    print 'func 函数内的 globals:',globals().keys()
    print 'func 函数内的 locals:',locals().keys()
print '__mian__函数内的 globals:',globals().keys()
print '__mian__函数内的 locals:',locals().keys()
func()
```

程序运行结果如下：

```
__mian__函数内的 globals: ['__builtins__', '__file__', '__package__', 'func', '__name__',
'__doc__']
__mian__函数内的 locals: ['__builtins__', '__file__', '__package__', 'func', '__name__',
'__doc__']
调用 func 函数
func 函数内的 globals: ['__builtins__', '__file__', '__package__', 'func', '__name__', '__
doc__']
func 函数内的 locals: ['astring', 'anum']
```

3. reload()

reload()函数可以重新导入一个已经导入的模块，其语法如下：

```
reload(module)
```

module 是用户想要重新导入的模块。使用该函数时，模块必须是全部导入，而不是通过 from-import 部分导入，而且它已成功被导入。此外，reload()函数的参数必须是模块自身而不是模块名称的字符串。

8.3 包

8.3.1 包的概述

包是一个有层次的文件目录结构，它定义了一个由模块和子包组成的 Python 应用程序执行环境。包可以解决如下问题：

(1) 把命名空间组织成有层次的结构；

(2) 允许程序员把有联系的模块组合到一起；

(3) 允许程序员使用有目录结构而不是一大堆杂乱无章的文件；

(4) 解决有冲突的模块名称。

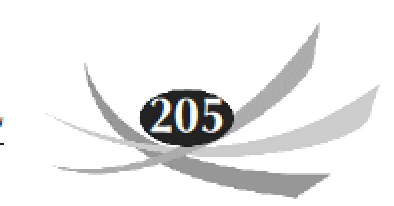
与模块相同，包也是使用句点属性标识来访问它们的元素。使用 import 语句和 from-import 语句都可以导入包中的模块。

假设一个包的目录结构如下：

```
Computer/
    __init__.py
    showComputerInfo.py
    MainEngine/
        __init__.py
        showMainEngineInfo.py
        MotherBoard/
            __init__.py
            showMotherBoardInfo.py
        RandomAccessMemory/
            __init__.py
            showRandomAccessMemoryInfo.py
        HardDisk/
            __init__.py
            showHardDiskInfo.py
    Monitor/
        __init__.py
        showMonitorInfo.py
```

Computer 是最顶级的包，MainEngine 和 Monitor 是它的一级子包，MotherBoard、Random AccessMemory 和 HardDisk 是它的二级子包，同时也是 MainEngine 包的一级子包。如果要导入 showHardDiskInfo 模块，可以通过以下方式导入：

```
#第一种方式
import Computer.MainEngine.HardDisk.showHardDiskInfo
Computer.MainEngine.HardDisk.showHardDiskInfo.MainFunction()
#第二种方式
from Computer import MainEngine.HardDisk.showHardDiskInfo
```




```
MainEngine.HardDisk.showHardDiskInfo.MainFunction()
#第三种方式
from Computer.MainEngine import HardDisk.showHardDiskInfo
HardDisk.showHardDiskInfo.MainFunction()
#第四种方式
from Computer.MainEngine.HardDisk import showHardDiskInfo
showHardDiskInfo.MainFunction()
#第五种方式
from Computer.MainEngine.HardDisk.showHardDiskInfo import MainFunction
MainFunction()
```

在上面的包目录结构中，可以看到很多__init__.py文件。这些文件是用来初始化相应模块的，通过from-import语句导入子包时需要用到该文件。如果不是通过这种方式导入的，这个文件可以是空文件。

8.3.2 包管理工具——pip

pip是一个用来管理Python包的工具，我们可以利用它来帮助我们安装、升级、卸载Python的第三方扩展包。特别是在Linux下，pip可以很轻松地做到自动化运维。

1. 安装pip

首先需要到https://pip.pypa.io/en/latest/installing.html站点下载get-pip.py文件，并将其保存到Python的安装根目录下。然后打开DOS命令窗口，用cd命令转到Python的安装根目录。本机把Python安装到C盘的Python27目录下，所以执行以下命令：

```
cd \Python27
```

其中反斜杠表示返回根目录（这里是C盘）。然后执行以下命令安装pig包管理工具：

```
python get-pip.py
```

安装成功后如图8-1所示。

安装完pip之后，我们可以访问在Python的安装根目录下的Scripts文件夹：

```
C:\Python27\Scripts
C:\Python27\Scripts\pip.exe
C:\Python27\Scripts\pip2.7.exe
C:\Python27\Scripts\pip2.exe
C:\Python27\Scripts\wheel.exe
```

2. 配置环境变量

打开Path系统变量，确保路径C:\Python27\Scripts已添加到Path环境变量中，如图8-2所示。

3. 管理包（安装、升级、卸载）

配置了环境变量之后，我们就可以在DOS命令窗口运行pip命令对包进行管理

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\wangs>cd \Python27

C:\Python27>python get-pip.py
Collecting pip
  Downloading pip-8.0.2-py2.py3-none-any.whl (1.2MB)
    100% |################################| 1.2MB 280kB/s
Collecting wheel
  Downloading wheel-0.26.0-py2.py3-none-any.whl (63kB)
    100% |################################| 65kB 1.1MB/s
Installing collected packages: pip, wheel
  Found existing installation: pip 7.0.1
    Uninstalling pip-7.0.1:
      Successfully uninstalled pip-7.0.1
Successfully installed pip-8.0.2 wheel-0.26.0

C:\Python27>_
```

图 8-1　安装 pip 包管理工具

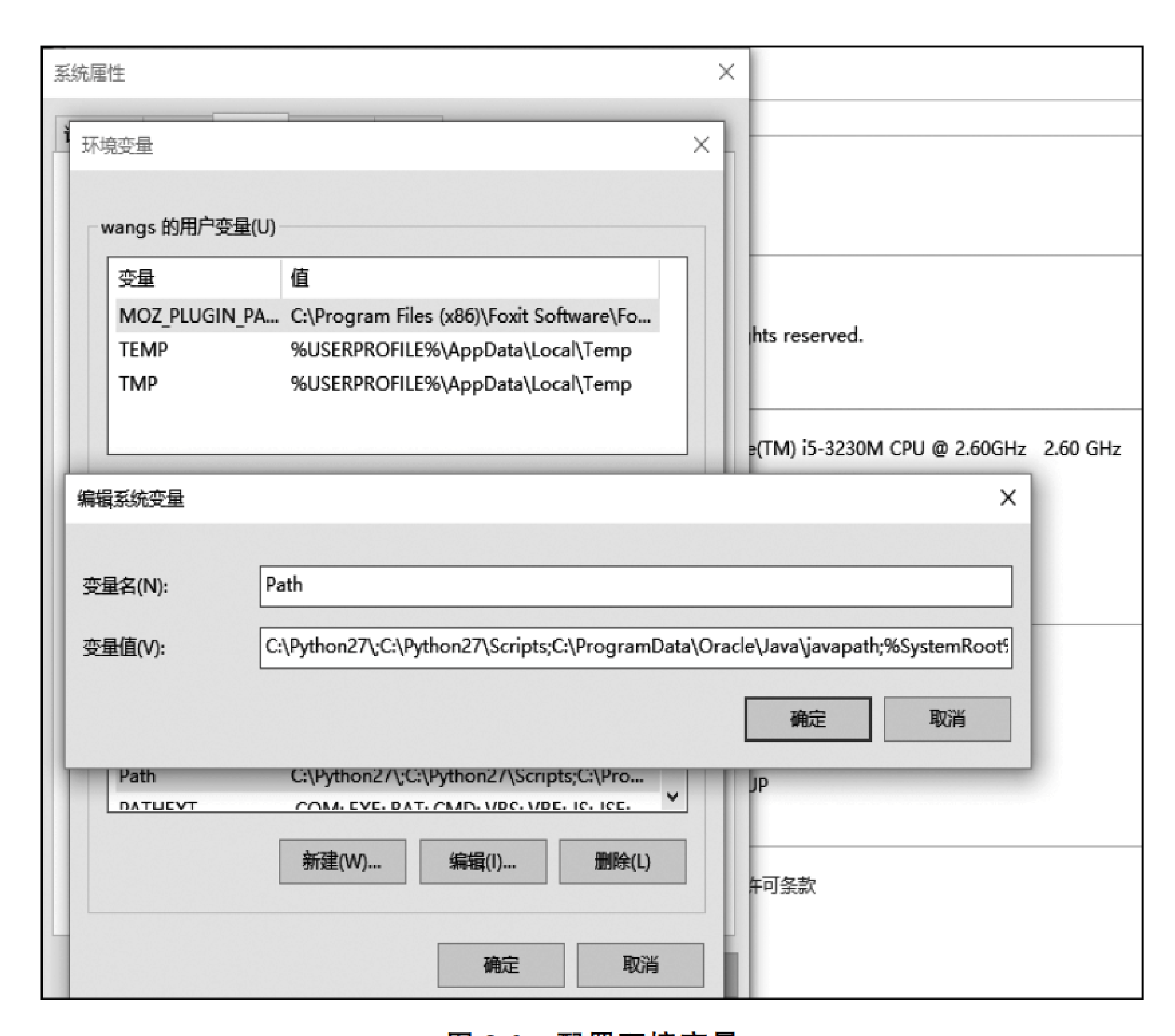


图 8-2　配置环境变量

了，如：

```
pip install SomePackage                      #安装一个包
pip install --upgrade SomePackage            #升级一个包
```

```
pip uninstall SomePackage                              #卸载一个包
```

8.4 本章小结

本章主要讲解了以下几个知识点：

(1) 命名空间。命名空间是从名称到对象的映射。Python 中有三类命名空间：内建命名空间、全局命名空间和局部命名空间。在不同的命名空间中的名称是没有关联的。此外，不同的全局命名空间或不同的局部命名空间，它们的名称也是没有关联的。命名空间都是动态创建的，并且每一个命名空间的生存时间也不一样。关于命名空间的三条规则如下：第一，赋值语句(包括显式赋值和隐式赋值)会把名称绑定到指定对象中，赋值的地方决定名称所处的命名空间；第二，函数、类定义会创建新的命名空间；第三，Python 搜索一个名称的顺序是“LEGB”。

(2) 模块。模块是把一组相关的名称、函数、类或者是它们的组合组织到一个文件中。一个文件可以被看作一个独立的模块，一个模块也可以被看作一个文件。模块的文件名就是模块的名字加上扩展名.py。

(3) 模块导入。模块导入可以使用 import 语句导入整个模块或者使用 from-import 语句导入指定模块的变量、函数或者类等。此外，当导入的模块或是模块属性名称已经在我们的程序中使用了，又或者是因为其名称太长，我们不想使用导入的名字，可以使用扩展的 import 语句(as)来解决这个问题。模块可以被导入多次，但只有第一次导入被加载并执行。模块导入主要有三个特性：载入时执行模块、导入与加载以及__future__特性。

(4) 模块内建函数。__import__()作为导入模块的函数是在执行 import(包括 from-import 和扩展 import)语句时调用的，提供这个函数是为了让有特殊需要的用户可以覆盖它，实现自定义的导入算法。globals()和 locals()内建函数分别返回调用处可以访问的全局和局部命名空间中的名称组成的字典。reload()函数可以重新导入一个已经导入的模块。

(5) 包。包是一个有层次的文件目录结构，它定义了一个由模块和子包组成的 Python 应用程序执行环境。与模块相同，包也是使用句点属性标识来访问它们的元素。使用 import 语句和 from-import 语句都可以导入包中的模块。可以使用包管理工具 pip 对包进行管理。

8.5 习　　题

一、解答题

1. 什么是命名空间？命名空间可以分为哪几类？命名空间有哪些规则？
2. 什么是模块？模块和文件有什么联系？
3. 导入一个模块有哪些方式？
4. 与模块相关的内建函数有哪些？它们的作用分别是什么？

5. 什么是包？如何导入包？

二、看程序写结果

1. 两个文件：module1.py 和 module2.py

module1.py 如下：

```
#module1.py
astring='abc'
def display():
    print 'astring from module1:',astring
```

module2.py 如下：

```
#module2.py
import module1 as m
m.display()
astring='123'
print 'astring from module2:',astring
m.display()
print '---------------------------------'
m.display()
m.astring='123'
print 'astring from module2:',astring
m.display()
```

现在执行 module2.py 脚本，其输出结果是什么？

2. 两个文件：module3.py 和 module4.py

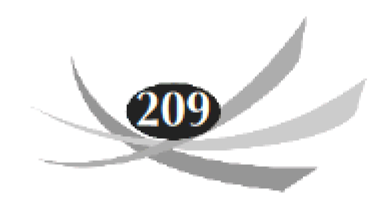
module3.py 如下：

```
#module3.py
class Reference:
    count=0
    def __init__(self):
        Reference.count+=1
        print 'count:',Reference.count
```

module4.py 如下：

```
#module4.py
import module3 as m1
r1=m1.Reference()
import module3 as m2
r2=m1.Reference()
r3=m2.Reference()
```

现在执行 module2.py 脚本，其输出结果是什么？



三、上机练习

将第 7 章上机练习的第 5 题组织成一个包结构，即电脑包是顶级包，它的一级子包是主机包，该包下还有一个显示器模块，而主机包下有主板模块、内存模块和硬盘模块。创建一个测试文件，通过导入上述的包或模块实现题目的要求。

第 9 章 异　常

本章学习目标

- 理解异常相关概念
- 掌握异常捕获的方式
- 掌握抛出异常和自定义异常
- 掌握 PythonWin 的调试方法
- 掌握 Eclipse for Python 的调试方法

在前面章节的例子中我们已经接触到了异常。当程序发生异常时，会提示程序出现什么错误，在修复指定错误之后，程序又可以正常运行了。本章将主要介绍什么是异常，如何捕获和根据需要创建自定义异常，此外，还会介绍如何用 PythonWin 和 Eclipse for Python 这两个常用开发工具来调试程序。

9.1 异　常

9.1.1 什么是异常

在介绍异常之前，我们先看看什么是错误。从软件方面来看，错误可以分为语法上的错误和逻辑上的错误。语法上的错误就是所编写的代码不符合指定语言的规范，导致不能被解析器解析或者编译器编译。这些错误必须在程序执行前修复。例如：

```
#coding:utf-8
#例 9-1
a=1
if a==1
    print 'the value of a is 1'
else
    print 'the value of a is not 1'
```

将其保存成 error. py，作为脚本在命令行上执行该程序，运行结果如下：

```
C:\Users\wangsd>C:\Python27\error.py
  File "C:\Python27\error.py", line 2
    if a==1
```

```
        ^
SyntaxError: invalid syntax
```

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的“箭头”，错误是由箭头前面的标记引起的。在这个例子中，检测到错误发生在 if 语句的条件，因为在它之后缺少一个冒号。可以看到，文件名和错误行号会一并输出，所以如果运行的是一个脚本我们就知道去哪里检查错误了。

当一个程序的语法正确，但运行时程序还会出错，这类错误就是逻辑错误了。逻辑错误可能是由于不完整、不合法的输入或者是做除法运算时，除数为零等原因导致的。

异常是程序运行过程中由于出现语法错误或逻辑错误而发生的事件，该事件可以中断程序指令的正常执行流程。在 Python 中，异常也是一个类，所有其他的异常都继承自 Exception 类，并且这个 Exception 类在 exceptions 模块中定义。Python 把所有的异常名称都放在内建命名空间中，以便程序无须导入 exceptions 模块就可以使用这些异常。

此外，还有两类异常 SystemExit 和 KeyboardInterrupt 不是由于程序出错导致的，System Exit 是由于当前 Python 应用程序需要退出，而 KeyboardInterrupt 代表用户按了 Ctrl+C 快捷键，想要关闭 Python。

注意：在新式类中，作为所有异常类的基类并不是 Exception，而是 BaseException。BaseException 有三个直接子类（派生类），分别是 SystemExit、KeyboardInterrupt 和 Exception。Exception 类的所有派生类又称为常规错误异常类，而所有系统提供的异常类统称为标准异常类。

9.1.2　标准异常类

如果程序出现错误，Python 会引发标准异常，提示程序的哪一行出现错误，并且还指出什么错误。Python(2.5 及以后的版本)中的标准异常类如表 9-1 所示。

表 9-1　Python 标准异常类

异常名称	说　　明
BaseException	所有异常类的基类
SystemExit	Python 解析器请求退出
KeyboardInterrupt	用户按了 Ctrl+C 快捷键，中断程序
Exception	常规错误异常类的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
StandardError	所有的内建标准异常类的基类
ArithmeticError	所有数值计算错误类的基类
FloatingPointError	浮点计算错误类
OverFlowError	数值计算超出最大限制

续表

异常名称	说　明
ZeroDivisionError	所有数据类型的除(或取模)零
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达 EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	Windows 系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有次索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误
NameError	未声明/初始化对象(没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(weak reference)试图访问已经被回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解析器系统错误
TypeError	对类型进行无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时的错误
UnicodeTranslateError	Unicode 转换时的错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告

续表

异 常 名 称	说　　明
OverflowWarning	关于自动提升为长整型的警告
PendingDeprecationWarning	关于特性将被废弃的警告
RuntimeWarning	运行时的警告
SystaxWarning	语法的警告
UserWarning	用户代码生成的警告

下面举例说明其中一些常见的异常。

1. ZeroDivisionError：除数为零

```
>>>1/0

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in<module>
    1/0
ZeroDivisionError: integer division or modulo by zero
```

这个例子我们使用的是整数除零，实际上，任何数值被零除都会抛 ZeroDivisionError 异常，提示除数为零。

2. AttributeError：尝试访问未知的对象属性

```
>>>class TestClass(object):
    a=1



>>>tc=TestClass()
>>>print tc.a
1
>>>print tc.b

Traceback (most recent call last):
  File "<pyshell#15>", line 1, in<module>
    print tc.b
AttributeError: 'TestClass' object has no attribute 'b'
```

在这个例子中，首先定义了一个类，类中包含一个类属性 a，然后创建类的实例化对象，通过 print 语句输出类属性 a，可以正常输出(1)，当试图输出不存在的属性 b 时，就抛 AttributeError 异常，提示 TestClass 类对象没有属性 b。

3. IOError：输入/输出错误

```
>>>f=open('a')
```

```
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in<module>
    f=open('a')
IOError: [Errno 2] No such file or directory: 'a'
```

在这个例子中,我们使用 open 内建函数试图打开一个不存在的文件 a,程序就抛 IOError 异常,提示没有 a 文件或没有 a 文件夹。

类似尝试打开一个不存在的磁盘文件一类的操作都会引发一个操作系统输入/输出(I/O)错误。而任何类型的 I/O 错误都会引发 IOError 异常。

4. IndexError:请求的索引超出序列范围

```
>>>aList=[1,2,3]
>>>print aList[3]

Traceback (most recent call last):
  File "<pyshell#18>", line 1, in<module>
    print aList[3]
IndexError: list index out of range
```

这个例子定义了一个包含三个元素的列表,然后尝试访问索引为 3 的元素,显然超出了该列表的索引范围,所以就引发了 IndexError 异常。

5. KeyError:请求一个不存在的字典关键字

```
>>>aDict={'book':'Python 编程基础','chapter':8}
>>>print aDict['book']
Python 编程基础
>>>print aDict['bookname']

Traceback (most recent call last):
  File "<pyshell#23>", line 1, in<module>
    print aDict['bookname']
KeyError: 'bookname'
```

这个例子定义了一个包含两个元素的字典,先访问存在的关键字 book 对应的值,可以正常输出(Python 编程基础),然后尝试访问不存在的关键字 bookname,结果就会引发 KeyError 异常。

6. NameError:尝试访问一个未声明的变量

```
>>>print astring

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in<module>
    print astring
NameError: name 'astring' is not defined
```

这个例子试图输出一个未声明的变量 astring,从而引发了 NameError 异常。任何可

以访问的变量都必须在命名空间中列出，访问变量时需要由解析器进行搜索，如果请求的名字在所有的命名空间都找不到，就会引发 NameError 异常。

7. SyntaxError：Python 解析器语法错误

本章开头的例子例 9-1 就引发了 SyntaxError 异常。SyntaxError 异常是唯一不是在运行时发生的异常。它代表 Python 代码中有一个不正确的结构，在改正之前程序无法执行。这些错误一般都是在编译时发生的，Python 解析器无法把脚本编译为 Python 字节码文件。

9.2 异常处理

当程序发生异常时，如果不捕获这些异常，然后对这些异常进行处理，程序就会终止执行。下面介绍几种捕获、处理异常的语句。

9.2.1 try...except 语句

程序中的异常，可以使用 try...except 语句来捕获，把需要执行的代码放到 try 语句块中，而把出现异常后的代码放到 except 语句块中。当 try 语句块中的代码出现异常后，会执行 except 语句块中的代码，从而捕获异常，进而根据异常做出相应的处理。try...except 语句结构的格式如下：

```
try:
    <需要执行的程序代码>
except 异常类 1 [,异常信息变量]:
    <当出现异常类 1所示的异常时执行的代码>
except (异常类 2, 异常类 3, 异常类 4, ……) [, 异常信息变量]:
    <当出现异常类 2、异常类 3、异常类 4……所示的异常时执行的代码>
```

其中，except 子句可以有多个，每个 except 子句也可以有多个异常类，并且通常都有异常信息变量，以便把异常信息保存到该变量中，然后在 except 子句中对该变量进行操作(通常是输出其异常信息)。

try...except 语句的异常处理规则如下：

- 执行 try 语句块中的语句，如果引发异常，则执行流程会跳到第一个 except 子句中。
- 如果第一个 except 子句中定义的异常与引发的异常匹配，则执行该 except 语句块中的语句。如果引发的异常与第一个 except 子句中的异常都不匹配，则会搜索第二个 except 子句，以此类推，直到搜索到与 try 语句块中引发的异常匹配的 except 子句为止，从而执行对应的 except 语句块中的语句。然后跳过后面的 except 子句(如果还有)执行后面的语句。
- 允许编写无限个 except 子句。为了使代码简洁，可以使用一个 except 子句，在该子句中把异常定义为 Exception，从而捕获所有引发的异常。
- 如果所有的 except 子句都不匹配，则异常会传递到上一个调用本代码的最高层 try 语句块中。

下面通过一个例子来理解 try...except 语句：

```
#coding:utf-8
#例 9-2  try...except 语句
def tryExcept():
    try:
        aList=[1,2,3]
        print 'aList[1]=',aList[1]
        print 'aList[3]=',aList[3]
    except NameError,e:
        #该语句主要是给用户看的,以增强程序的用户体验性
        print 'There is a NameError'
        #该语句主要是给程序员看的,以便程序员能尽快修复程序
        print 'The reason of occurring exception is',e
    except KeyError,e:
        print 'There is a KeyError'
        print 'The reason of occurring exception is',e
    except IndexError,e:
        print 'There is a IndexError'
        print 'The reason of occurring exception is',e
    except IOError,e:
        print 'There is a IOError'
        print 'The reason of occurring exception is',e

tryExcept()
```

在 try 语句块中，首先定义了一个包含三个元素的列表 aList，然后访问索引为 1 的元素，可以正常输出，接着访问索引为 3 的元素，显然超出了列表 aList 的索引范围。因此引发 IndexError 异常，程序跳到第一个 except 子句中，检测其是否含有 IndexError，没有，然后程序跳到第二个 except 子句，继续检测其是否含有 IndexError，若还是没有，程序继续往下，跳到第三个 except 子句，发现其含有 IndexError，即与 try 语句块中引发的异常相匹配，则执行该语句块中的代码，然后跳过后面的 except 子句(except IOError，e)，执行下一条语句。运行该程序，输出结果如下：

```
aList[1]=2
aList[3]=There is a IndexError
The reason of occurring exception is list index out of range
```

程序中的每个 except 子句后都有一个变量 e，当程序执行到与之匹配的 except 子句时，把在 try 引发异常的原因信息保存到该变量中，然后，在匹配的 except 语句块中输出该变量，以提示编程人员程序出现异常的原因。

当然，也可以把程序中的多个 except 子句改为一个 except 子句，然后在该子句列出可能的异常，如：

```
def tryExcept():
```

```
    try:
        aList=[1,2,3]
        print 'aList[1]=',aList[1]
        print 'aList[3]=',aList[3]
    except (NameError,KeyError,IndexError,IOError),e:
        print 'There is a Error'
        print 'The reason of occurring exception is',e

tryExcept()
```

但是，这种方式不能根据不同的异常给出不同的响应，而只能给出一个统一的响应。

如果需要捕获程序中发生的所有异常，只需在一个 except 子句中把异常类指定为 Exception（对于经典类）或 BaseException（对于新式类），这样，当程序出现异常时，该 except 子句都会被执行，如：

```
def tryExcept():
    try:
        aList=[1,2,3]
        print 'aList[1]=',aList[1]
        print 'aList[3]=',aList[3]
    except (Exception),e:
        print 'There is a Error'
        print 'The reason of occurring exception is',e

tryExcept()
```

甚至还可以是空的 except 子句，但是，这种方式无法直接获取引发异常的原因，因为 except 子句没有任何的异常类，也就不能为其添加一个用于保存引发异常原因信息的变量。

9.2.2　try...except...else 语句

在前面的章节中，我们介绍了 else 子句可以和条件语句或循环语句组合起来一起使用。本小节将介绍另一种与 else 子句组合的语句 try...except，即 try...except...else 语句。该语句可以用来捕获异常，使用 try...except...else 语句捕获异常与使用 try...except 语句捕获异常类似，区别在于其多了一个 else 子句。当 try 语句块中没有发生异常时，程序会跳过所有的 except 子句，执行 else 子句中的代码。try...except...else 语句结构的格式如下：

```
try:
    <需要执行的程序代码>
except 异常类 1 [,异常信息变量]:
    <当出现异常类 1 所示的异常时执行的代码>
except (异常类 2, 异常类 3, 异常类 4, ……) [, 异常信息变量]:
    <当出现异常类 2、异常类 3、异常类 4……所示的异常时执行的代码>
```

```
……
else:
    <当程序没有发生异常时执行的代码>
```

try...except...else 语句的执行规则与 try...except 语句的大同小异,其具体规则如下:

- 执行 try 语句块中的语句,如果引发异常,则执行流程会跳到第一个 except 子句中。
- 如果程序找到与 try 语句块中引发的异常匹配的 except 子句,则执行对应的 except 语句块中的语句。然后跳过后面的 except 子句(如果还有)和 else 子句,执行后面的语句。
- 如果所有的 except 子句都不匹配,则异常会传递到上一个调用本代码的最高层 try 语句块中。
- 如果执行 try 语句块中的所有语句都没有发生异常,则程序会执行 else 子句中的代码。

简言之,except 子句会捕获 try 语句块发生的异常(假定有与之匹配的异常),而 else 子句只在 try 语句块中没有发生异常,并且没有 return 语句时执行。

注意:如果 try 语句块没有发生异常,但是有 return 语句,则 else 子句是没有机会被执行的。

下面通过一个例子来理解 try...except...else 语句。

```
#coding:utf-8
#例 9-3  try...except...else 语句
def tryExceptElse():
    try:
        num1=2
        num2=1
        result=num1/num2
    except NameError,e:
        print 'There is a NameError'
        print 'The reason of occurring exception is',e
    except KeyError,e:
        print 'There is a KeyError'
        print 'The reason of occurring exception is',e
    except IndexError,e:
        print 'There is a IndexError'
        print 'The reason of occurring exception is',e
    except ZeroDivisionError,e:
        print 'There is a ZeroDivisionError'
        print 'The reason of occurring exception is',e
    else:
        print 'The program runs successfully'
```

```
        print '%d/%d=%d' % (num1,num2,result)
        return result

value=tryExceptElse()
print 'The return value is',value
```

显然,这个程序在执行 try 语句块中并不会发生异常,且在该语句块中没有 return 语句,因此,在执行完 try 语句块之后直接跳过 except 子句,而跳到 else 子句,然后执行 else 子句中的代码。运行该程序,输出结果如下:

```
The program runs successfully
2/1=2
The return value is 2
```

但是,当 try 语句块中存在 return 语句(把计算结果返回),则 else 子句就不会执行,输出结果如下:

```
The return value is 2
```

9.2.3 try...except...finally 语句

try...except 语句除了可以和 else 子句组合之外,还可以和 finally 子句组合,即 try...except... finally 语句。该语句也可以用来捕获异常,然而 try...except...finally 语句又和 try...except 语句或 try...except...else 语句不同,它使得 finally 子句中的代码肯定会被执行,而无论 try 语句块中是否发生异常。加入 finally 子句的目的是维护程序的一致性,如文件的关闭以及断开服务器的连接等。try...except...finally 语句结构的格式如下:

```
try:
    <需要执行的程序代码>
except 异常类 1 [,异常信息变量]:
    <当出现异常类 1 所示的异常时执行的代码>
except (异常类 2, 异常类 3, 异常类 4, ......) [, 异常信息变量]:
    <当出现异常类 2、异常类 3、异常类 4......所示的异常时执行的代码>
......
finally:
    <执行完 try 语句块中的代码之后要执行的代码>
```

try...except...finally 语句的执行规则如下:

- 执行 try 语句块中的语句,如果引发异常,则执行流程会跳到第一个 except 子句中。
- 如果程序找到与 try 语句块中引发的异常匹配的 except 子句,则执行对应的 except 语句块中的语句。然后跳过后面的 except 子句(如果还有),执行 finally 子句。
- 如果所有的 except 子句都不匹配,则先执行后面的 finally 子句,然后把异常传递到上一个调用本代码的最高层 try 语句块中。

- 如果执行 try 语句块中的所有语句都没有发生异常，程序同样会执行 finally 子句。

简言之，无论 try 语句块中是否发生异常，finally 子句中的代码都会被执行。

下面通过一个例子来理解 try...except...finally 语句。

```
#coding:utf-8
#例 9-4  try...except...finally语句
def tryExceptFinally():
    try:
        f=open('a')                                  #文件 a 不存在磁盘中
        print '打开文件'
    except NameError,e:
        print 'There is a NameError'
        print 'The reason of occurring exception is',e
    except KeyError,e:
        print 'There is a KeyError'
        print 'The reason of occurring exception is',e
    except IndexError,e:
        print 'There is a IndexError'
        print 'The reason of occurring exception is',e
    except IOError,e:
        print 'There is a IOError'
        print 'The reason of occurring exception is',e
    finally:
        print '关闭文件'

tryExceptFinally()
```

该程序在 try 语句块中尝试打开一个不存在磁盘的文件 a，因此会引发 IOError 异常，而最后一个 except 子句中的异常与所引发的 IOError 异常相匹配，然后执行该 except 子句的代码。接着会执行 finally 子句的代码。

注意：在 try 语句块中的某条语句引发异常后，try 语句块内该条语句后面的语句就不会执行，因此不会输出“打开文件”。运行该程序，输出结果如下：

```
There is a IOError
The reason of occurring exception is [Errno 2] No such file or directory: 'a'
关闭文件
```

此外，finally 子句可以仅与 try 子句组合，即 try...finally 语句。该语句和 try...except 语句的区别在于它不是用来捕获异常的，而是维护程序的一致性(关闭打开的文件和断开服务器的连接等)。如：

```
try:
    f=open('test.py')
    print '打开文件'
```

```
finally:
    f.close()
    print '文件已关闭'
```

实际上，try…except…finally 语句还可以与 else 子句组合，即 try…except…else…finally 语句。

总的来说，捕获异常的只有 except 子句，组合 else 子句或 finally 子句是为了丰富异常的处理方式，使程序更合理、稳定。

9.3　抛出异常和自定义异常

除了解析器自动抛出(引发)异常之外，Python 还允许程序员手动抛出异常。此外，除了内建的异常类型，Python 也允许程序员自定义所需的异常类型，用于描述 Python 内建异常类型中没有涉及的异常情况。下面将对抛出异常和自定义异常分别进行叙述。

9.3.1　抛出异常

到目前为止，我们所见到的异常都是由于程序执行期间发生错误而由解析器自动抛出(引发)的。程序员在编写程序时也希望在遇到错误的输入等原因时能够手动抛出异常，为此，Python 提供了 raise 关键字让程序员明确地抛出哪种异常，raise 语句的一般用法如下：

```
raise 异常类 [,异常参数]
```

其中，异常类既可以是内建的异常类型，也可以是自定义的异常类型(在下一小节介绍)，异常参数是可选的，这个参数可以是一个单独的对象，也可以是多对象的元组。当异常发生时，异常的参数总是作为一个元组传到异常处理器中。即如果异常参数是一个对象，就会生成只有一个元素的元组，而如果是元组，则直接传给异常处理器。

下面通过一个例子来理解使用 raise 语句手动抛出异常。

```
#coding:utf-8
#例 9-5　使用 raise 语句抛出内建异常
def throwException():
    try:
        userInfoDict={'admin':'admin'}
        username=raw_input('please enter an username:')
        password=raw_input('please enter a password:')
        if  username!='admin':
            raise KeyError,('throw an exception manually', 'line 7')
        elif userInfoDict[username]!=password:
            print 'password is not correct'
        else:
            print 'login successfully'
    except KeyError,e:
```

```
        print 'There is a KeyError'
        print e

throwException()
```

该程序首先定义了一个字典,key 和 value 都是 admin,然后提示用户输入用户名和密码,如果用户输入的用户名不是 admin,将通过 raise 语句引发一个 KeyError 异常,并指定异常参数值,然后在 except 子句中捕获 KeyError 异常,并把 raise 语句指定的异常参数值传给变量 e。如果输入的用户名和密码都是 123,则输出结果如下:

```
please enter an username:123
please enter a password:123
There is a KeyError
('throw an exception manually', 'line 7')
```

实际上,raise 语句更多地用于抛出用户自定义的异常类型。

9.3.2 自定义异常

当 Python 的内建异常类型不能满足我们的需要时,我们可以自定义异常的类型。自定义异常是一个类,它必须继承 Exception 或者 BaseException 类,按照命名规范,自定义异常通常以 Error 结尾,以显式地告诉程序员出现异常的类型,自定义异常只能通过 raise 语句手动抛出。

把例 9-5 抛出的 KeyError 异常改为自定义异常,修改后如下:

```
#coding:utf-8
#例 9-6  使用 raise 语句抛自定义异常
class UserInfoError(Exception):
    def __init__(self,code,message):
        self.code=code
        self.message=message

    def __str__(self):
        errorJsonInfo='{"code":"%d","message":"%s"}' % (self.code,self.message)
        return errorJsonInfo

def throwException():
    try:
        userInfoDict={'admin':'admin'}
        username=raw_input('please enter an username:')
        password=raw_input('please enter a password:')
        if  username!='admin':
            raise UserInfoError,UserInfoError(40001,'username is not correct')
        elif userInfoDict[username]!=password:
            raise UserInfoError,UserInfoError(40002,'password is not correct')
```

```
        else:
            print 'login successfully'
    except UserInfoError,e:
        print 'There is a UserInfoError'
        print e

throwException()
```

该程序首先定义一个异常类 UserInfoError,并指定其基类为 Exception。在__init__构造方法中初始化 code 属性和 message 属性,即保存错误的编号和错误的信息,在__str__方法中把错误信息封装成 json 格式的信息(json 格式是业界用来封装数据的常用格式),以便使用 print 语句时按 json 格式输出错误信息。

当用户输入用户名和密码后,首先判断用户名是否为 admin,如果不是,抛前面定义好的 UserInfoError 异常,并创建这个异常类的实例化对象,作为异常参数,提示用户名错误;如果是,再判断密码是否是用户字典 userInfoDict 中对应的密码,如果不是,同样抛 UserInfoError 异常,并把该异常类的实例化对象作为异常参数,提示密码错误;如果用户名和密码与用户字典 userInfoDict 中的一致,则输出登录成功信息。根据不同的输入,有不同的输出:

1. 用户名不是 admin,密码任意字符

```
please enter an username:123
please enter a password:123
There is a UserInfoError
{"code":"40001","message":"username is not correct"}
```

2. 用户名是 admin,密码不是 admin

```
please enter an username:admin
please enter a password:123
There is a UserInfoError
{"code":"40002","message":"password is not correct"}
```

3. 用户名和密码都是 admin

```
please enter an username:admin
please enter a password:admin
login successfully
```

9.4 调试程序

令程序员头疼的是程序出错了,但不知道具体是什么原因导致的,这就需要对程序进行调试。所谓程序调试,就是在将编制的应用程序投入实际运行前,以手工编译程序等方式进行测试,修正语法错误和逻辑错误的过程。这是保证应用程序正确性必不可少的步骤。无论是 Java 语言还是.NET 语言,都有相应的调试工具,Python 也不例外。下面介

绍两种用于调试 Python 程序的工具。

9.4.1 使用 PythonWin 调试程序

PythonWin 是一个优秀的 Python 集成开发环境，在许多方面都比 IDE 优秀。正如其文件名所示，这个工具是针对 Windows 用户的。下载地址：http://sourceforge.net/projects/pywin32 /files/pywin32/Build%20218/。

打开以上网址后如图 9-1 所示。

Looking for the latest version? **Download README.txt (951 Bytes)**

Home / pywin32 / Build 218

Name	Modified	Size	Downloads / Week
Parent folder			
pywin32-218.win-amd64-py3.3.exe	2012-10-29	8.6 MB	56
pywin32-218.win-amd64-py3.4.exe	2012-10-29	8.6 MB	63
pywin32-218.win-amd64-py3.2.exe	2012-10-29	7.3 MB	15
pywin32-218.win-amd64-py3.1.exe	2012-10-29	7.3 MB	1
pywin32-218.win-amd64-py2.7.exe	2012-10-29	7.3 MB	269
pywin32-218.win-amd64-py2.6.exe	2012-10-29	7.3 MB	5
pywin32-218.zip	2012-10-29	7.0 MB	44
pywin32-218.win32-py3.4.exe	2012-10-29	7.9 MB	52
pywin32-218.win32-py3.3.exe	2012-10-29	7.9 MB	12
README.txt	2012-10-29	1.6 kB	6

图 9-1 PythonWin 的下载网页

注意：要根据自己安装的 Python 版本选择对应的 PythonWin 版本。

安装 PythonWin 非常简单，默认安装即可，这里不再详述。下面集中介绍使用 PythonWin 调试程序。使用 PythonWin 调试程序可以分为以下几个步骤。

(1) 打开要调试的 Python 源程序文件。执行 File→Open 命令，将要调试 d 文件在 PythonWin 中打开。

(2) 设置断点。将鼠标放在可能出现错误的代码行，然后执行 File→Debug 命令，弹出含有许多选项的子菜单，其中 Go 表示开始调试，快捷键为 F5；Step in 表示单步执行，快捷键为 F11；Step out 表示单步跳出，快捷键为 F10；Stop 表示停止调试，快捷键为 Shift＋F5；Toggle Breakpoint 表示设置或取消断点，快捷键为 F9。在此，我们选择 Toggle Breakpoint 选项设置断点，如图 9-2 所示。

(3) 当设置完所有的断点后，在文件左侧会出现白色的圆圈，如图 9-3 所示。

(4) 设置断点后，按 F5 快捷键启动程序的调试模式，如图 9-4 所示。程序将弹出一个窗口，等待用户输入。

(5) 查看变量的值。单击 OK 按钮或者直接按回车键，程序将在第一个断点处停止

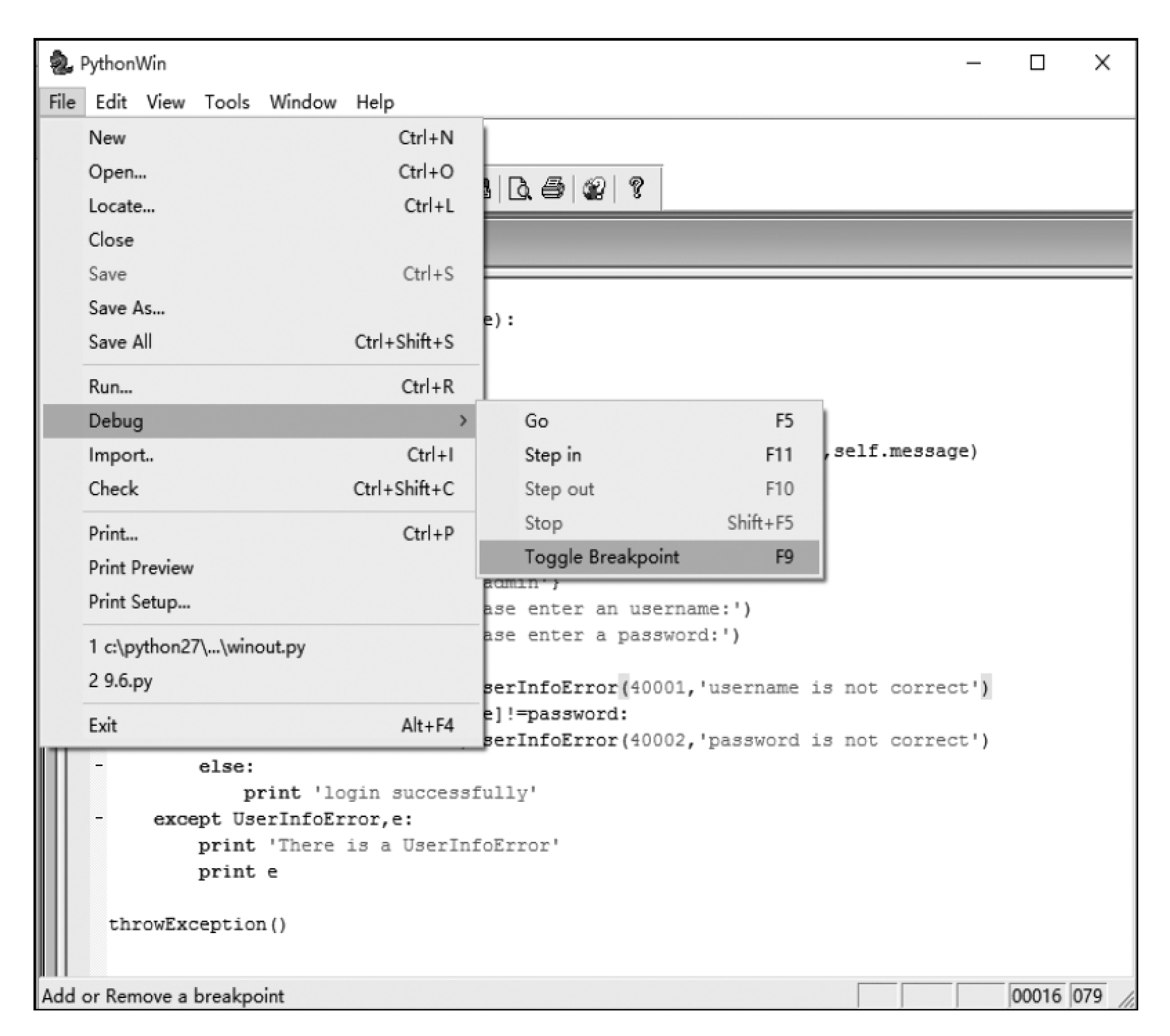


图 9-2　设置断点示意图

PythonWin

File Edit View Tools Window Help

9.6.py

```
class UserInfoError(Exception):
    def __init__(self,code,message):
        self.code = code
        self.message = message

    def __str__(self):
        errorinfo = '{"code":"%d","message":"%s"}' % (self.code,self.message)
        return errorinfo

def throwException():
    try:
        userInfoDict = {'admin':'admin'}
        username = raw_input('please enter an username:')
        password = raw_input('please enter a password:')
        if  username!='admin':
            raise UserInfoError,UserInfoError(40001,'username is not correct')
        elif userInfoDict[username]!=password:
            raise UserInfoError,UserInfoError(40002,'password is not correct')
        else:
            print 'login successfully'
    except UserInfoError,e:
        print 'There is a UserInfoError'
        print e

throwException()
```

Setting breakpoint　00023 016

图 9-3　设置断点后的示意图

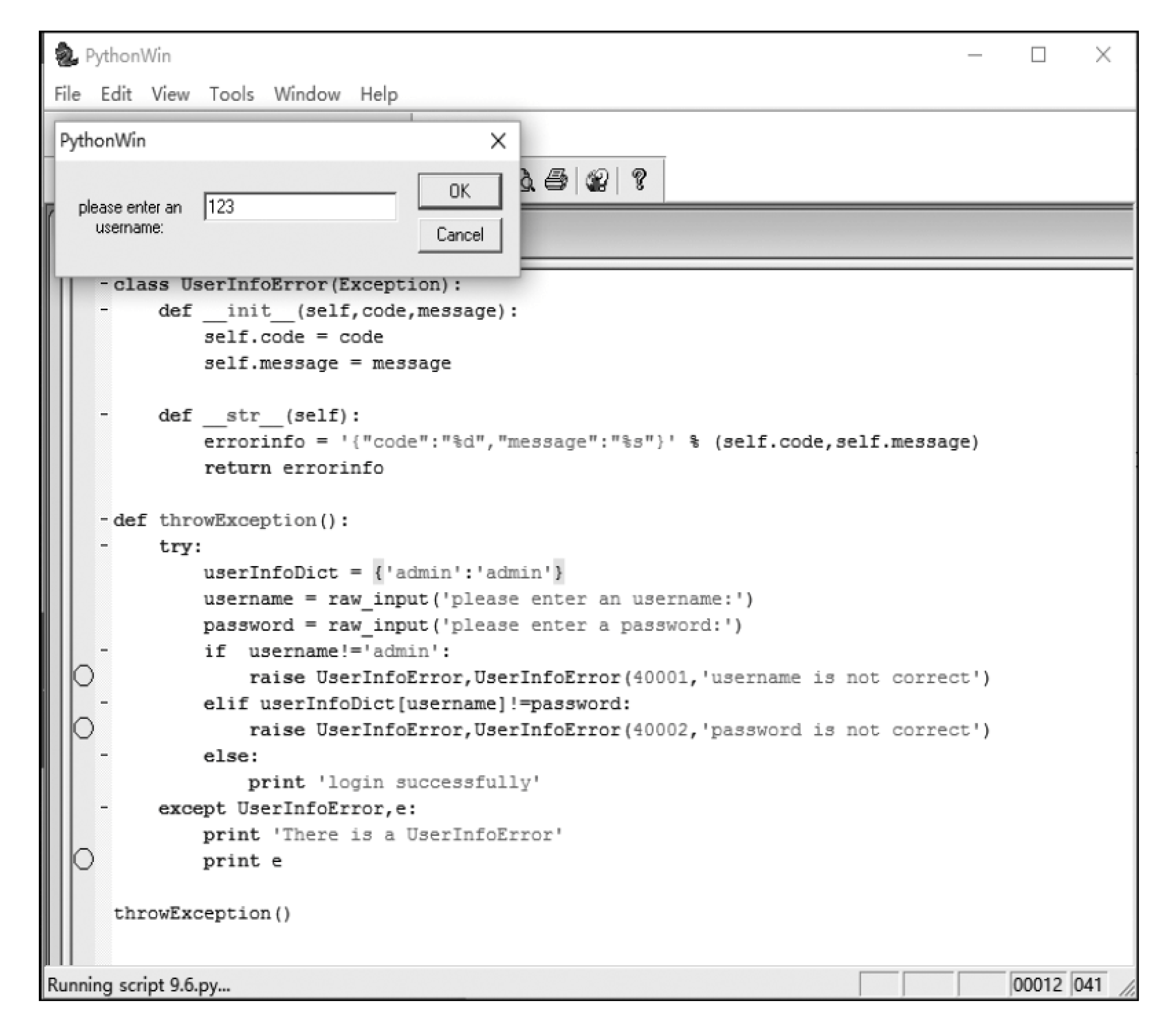


图 9-4　启动程序调试模式

执行,如图 9-5 所示。

```
PythonWin - break
File Edit View Tools Window Help
9.6.py
class UserInfoError(Exception):
    def __init__(self,code,message):
        self.code = code
        self.message = message

    def __str__(self):
        errorinfo = '{"code":"%d","message":"%s"}' % (self.code,self.message)
        return errorinfo

def throwException():
    try:
        userInfoDict = {'admin':'admin'}
        username = raw_input('please enter an username:')
        password = raw_input('please enter a password:')
        if username!='admin':
            raise UserInfoError,UserInfoError(40001,'username is not correct')
        elif userInfoDict[username]!=password:
            raise UserInfoError,UserInfoError(40002,'password is not correct')
        else:
            print 'login successfully'
    except UserInfoError,e:
        print 'There is a UserInfoError'
        print e

throwException()
Running script 9.6.py...                                    00016 001
```

图 9-5　启动程序的单步调试模式

(6) 查看变量的值。执行 View→Interactive Window 命令，弹出一个 Interactive Window 窗口，在这个窗口中，可以输入变量，按回车键后，在下一行可以看到对应的值。如图 9-6 所示。注意，只能查看程序运行到此行可以访问的变量。

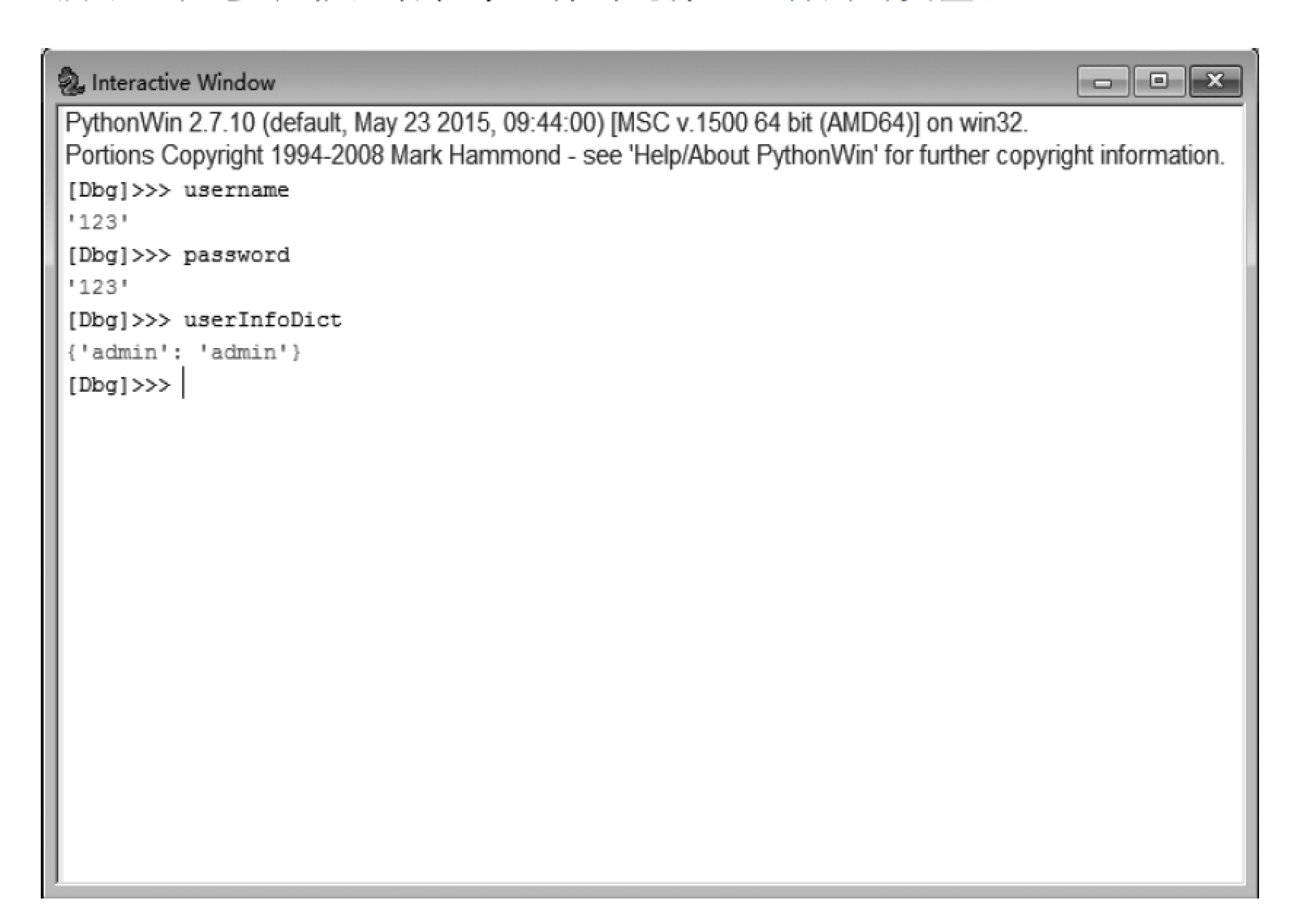


图 9-6　查看变量值

(7) 按 F5 键跳到下一个断点，我们假定在 except 子句中把 print e 误写成 print ee，然后按 F11 键执行 print ee 这行代码，这时在 Interactive Window 窗口中可以看到 NameError 异常，提示名称 ee 没有定义，如图 9-7 所示。

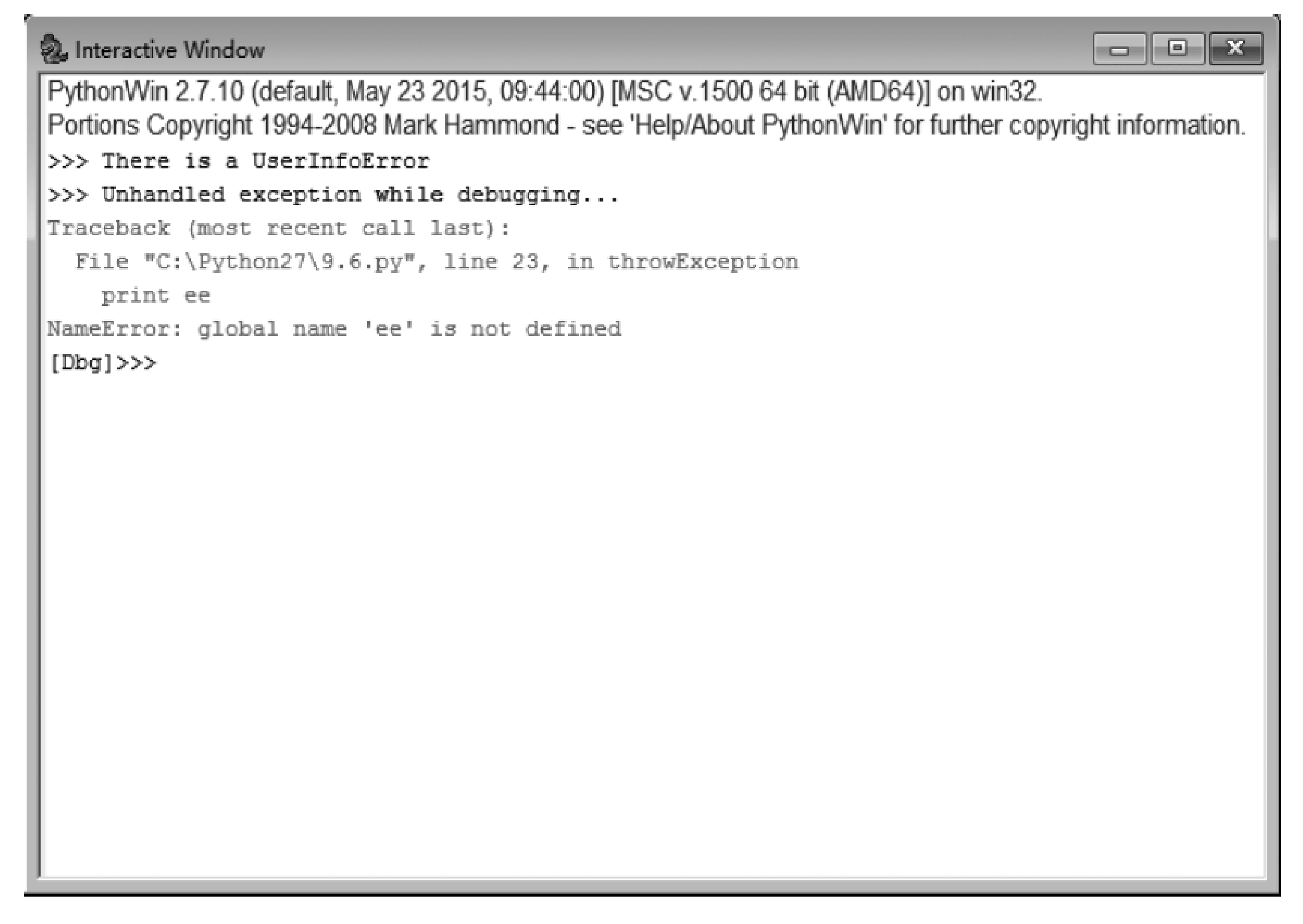


图 9-7　出现 NameError 异常

(8) 当发现出现异常所在的代码行后,按 Shift+F5 键停止调试,修改代码,然后再次运行程序。

9.4.2 使用 Eclipse for Python 调试程序

PythonWin 调试程序的操作与常用开发工具的使用习惯不同,它是在另一个窗口中才能查看程序总共的变量,使用起来不够方便。Eclipse 是一款功能非常强大的集成开发工具,它除了支持 C、Java、PHP 等语言外,还支持 Python 语言,并且提供调试功能(需要安装 PyDev 插件)。

Eclipse 无须安装,下载解压后即可(运行 Eclipse 前要确保计算机已安装 JDK(Java Development Kit))。

第一次运行 Eclipse 时会提示设置工作空间的路径,工作空间就是存放 Eclipse 工程的目录,如图 9-8 所示。

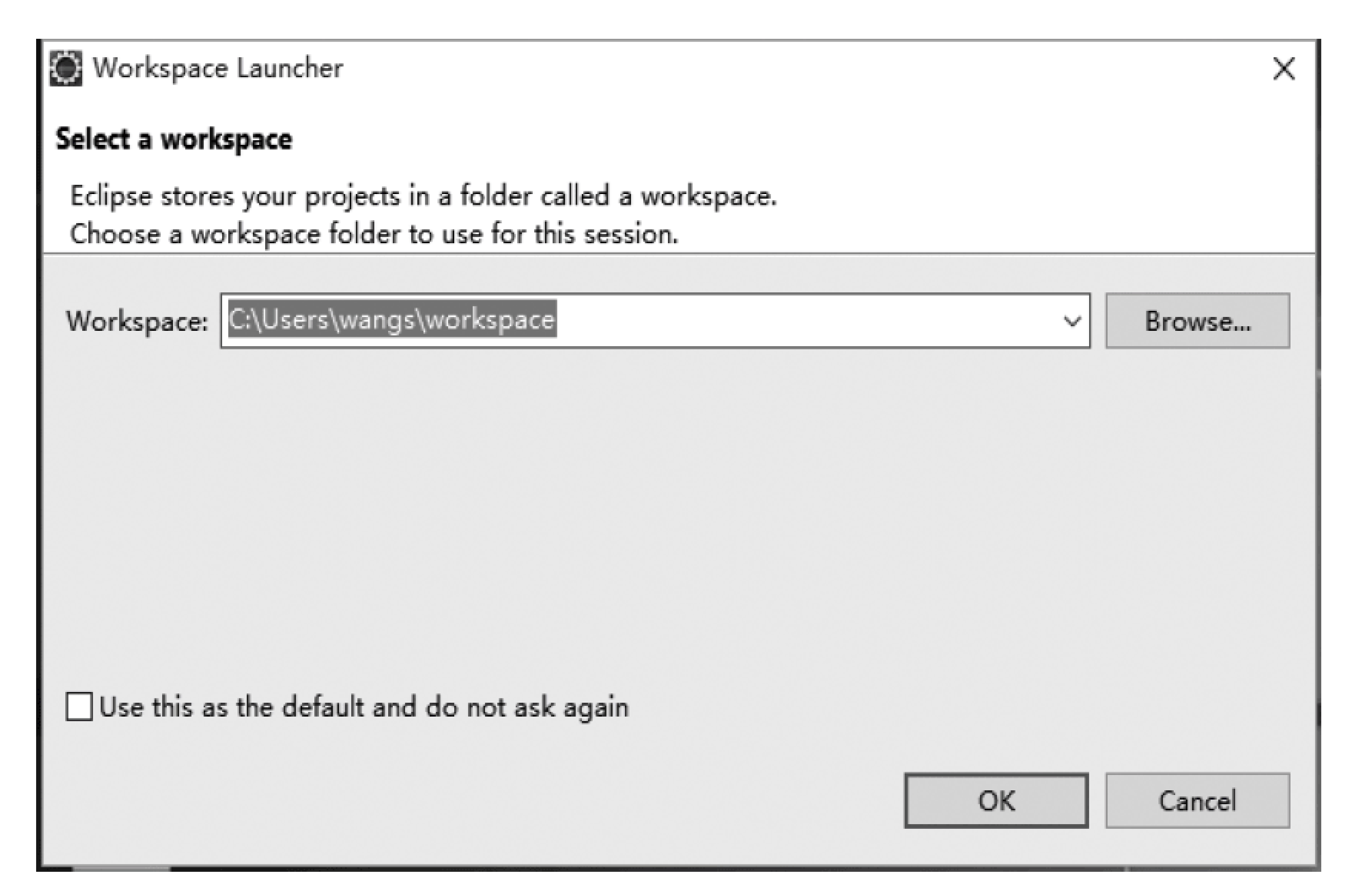


图 9-8　工作空间的路径设置

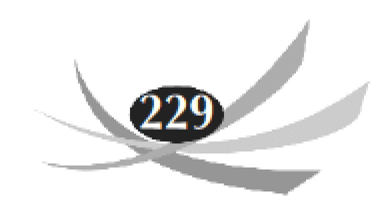
在这里,我们选择默认路径即可,然后勾选下面的复选框(Use this as the default and do not ask again),按 OK,下次运行 Eclipse 就不会再跳出这个窗口。

在调试之前需要在 Eclipse 中新建一个 Python 工程,而新建 Python 工程需要为 Eclipse 安装 PyDev 插件,安装步骤如下:

(1) 在浏览器中输入地址: http://marketplace.eclipse.org/node/114,打开后如图 9-9 所示。

(2) 把鼠标移到 Install 图标中,单击并拖曳其到 Eclipse 窗口中(运行 Eclipse 后默认显示的窗口)。然后弹出一个 Confirm Selected Features 对话框,如图 9-10 所示,单击 Comfirm 按钮。

(3) 在弹出的 Review Licenses 对话框中,选择右栏下的 I accept the terms of the license agreements,然后单击 Finish 按钮,如图 9-11 所示。



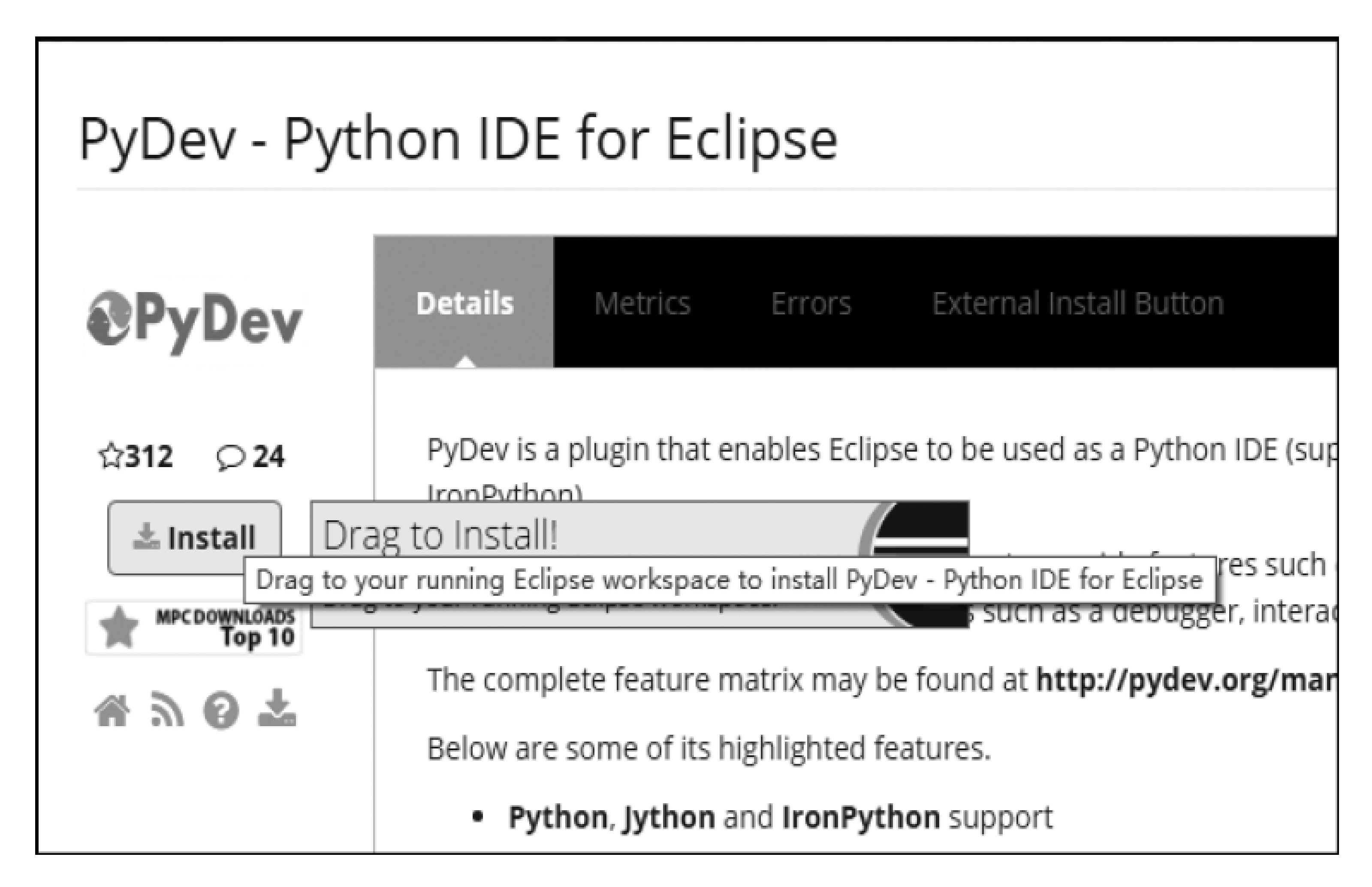


图 9-9 PyDev 插件的下载网页

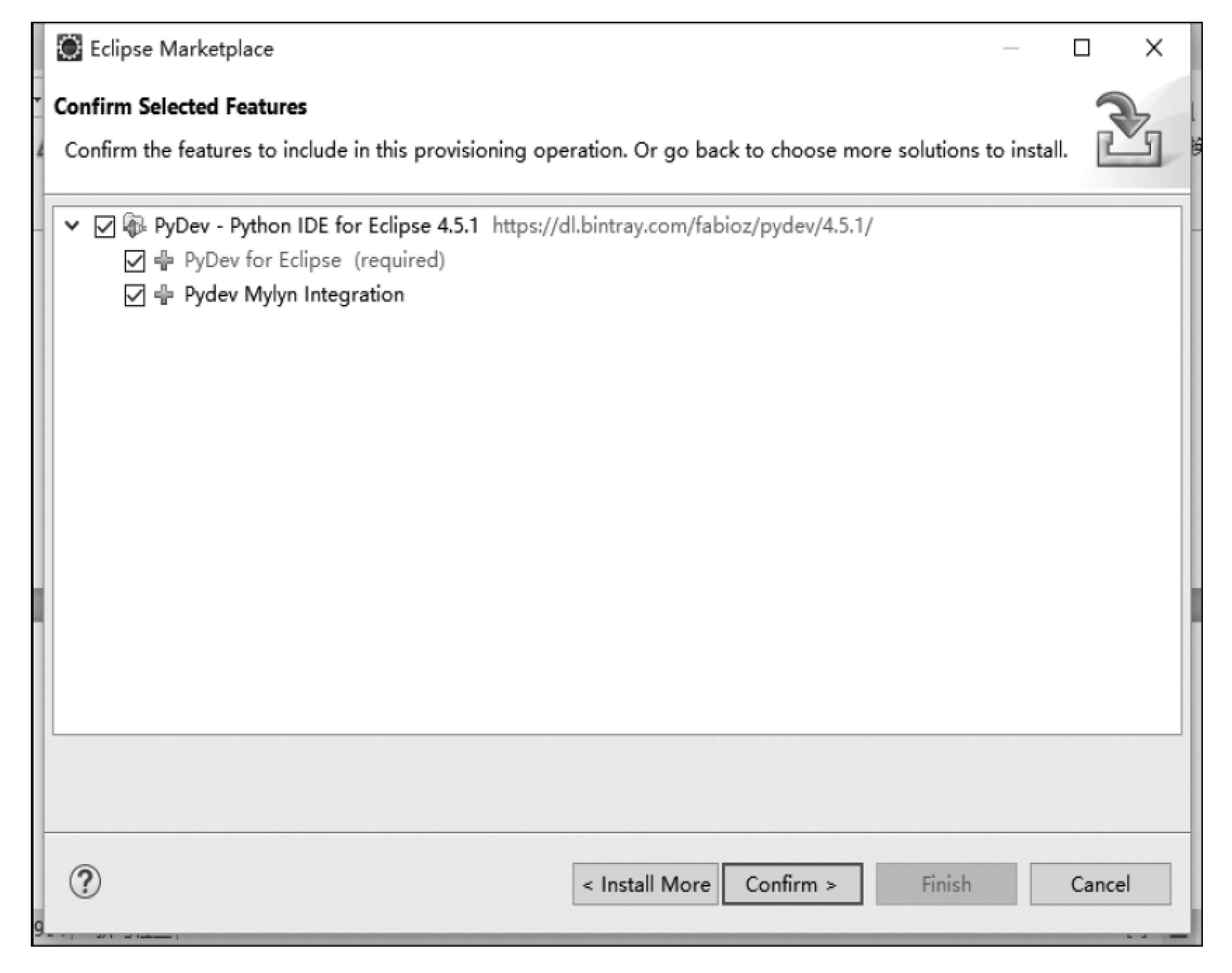


图 9-10 Confirm Selected Features 对话框

(4) 在弹出的 Selection Needed 对话框中，勾选下方的复选框，信任这些软件，再单击 OK 按钮，如图 9-12 所示。

(5) 安装成功后，会提示重启 Eclipse，如图 9-13 所示。

安装 PyDev 插件后，就可以在 Eclipse 中新建 Python 工程，然后对程序进行调试了。

1. 新建 Python 项目

第一次新建 Python 项目，执行 File→New→Project 命令，在弹出的 New Project 对话框中选择 PyDev 目录下的 PyDev Project，然后单击 Next 按钮，如图 9-14 所示。

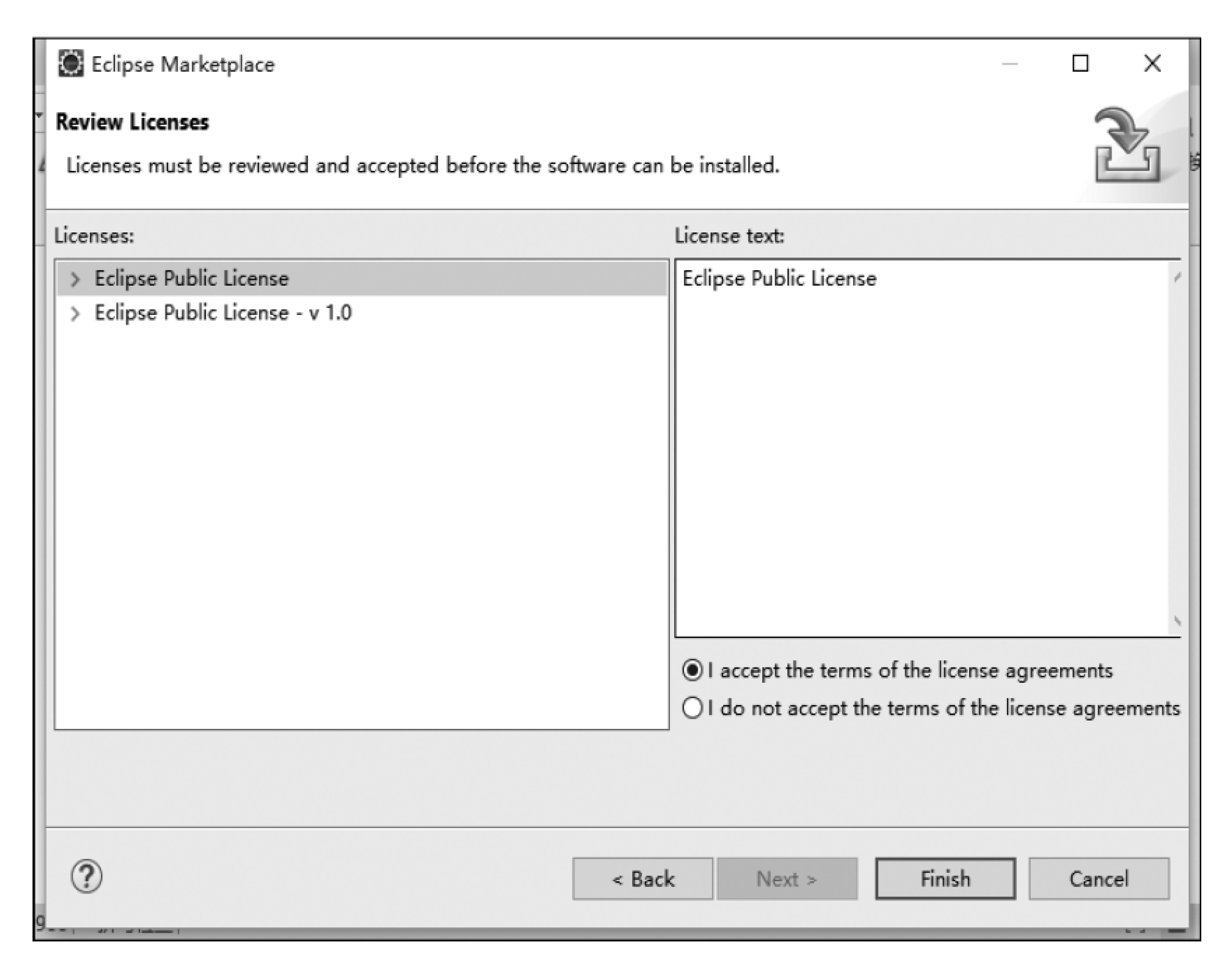


图 9-11　Review Licenses 对话框

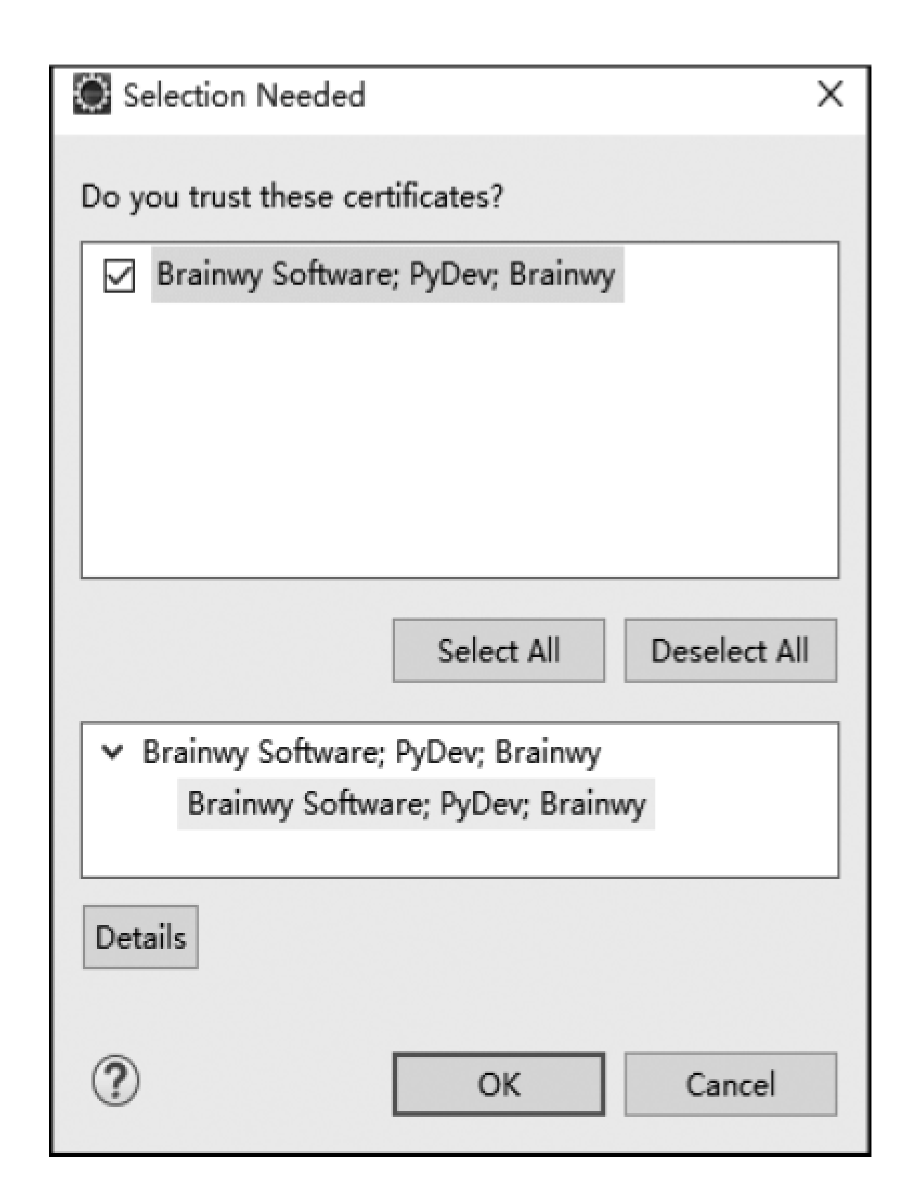


图 9-12　Selection Needed 对话框

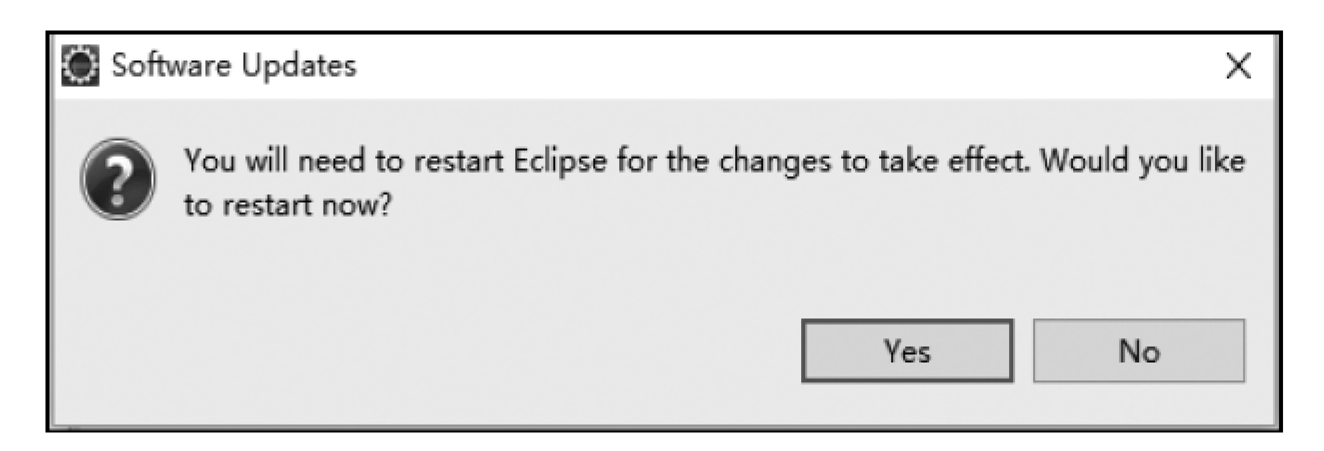


图 9-13　提示重启 Eclipse

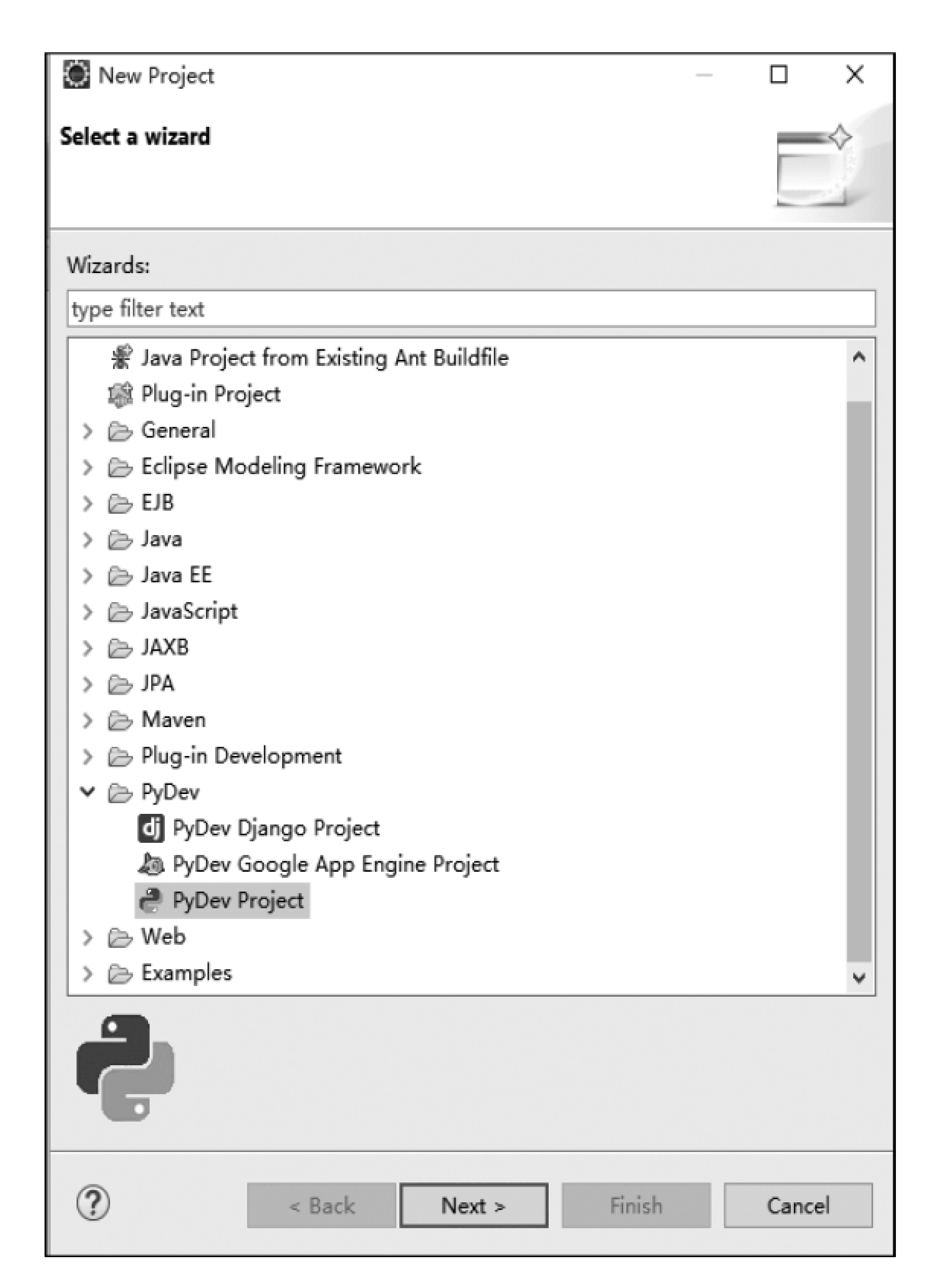


图 9-14　选择创建的工程

第一次新建 Python 项目成功后，下次再新建 Python 项目就可以直接执行 File→New→PyDev Project 命令，使其弹出 PyDev Project 对话框。在这个对话框中可以设置工程的属性，包括工程的名称、存储的位置、使用的 Python 标准库等信息。在名称为 Project name 的文本框中输入工程名称 PythonDemo，在 Project contents 区域中设置工程的存储路径，这里使用默认路径，如果要使用其他的存储路径，可以单击 Browse 按钮更改工程的存储路径。在 Python type 选项组中选择 Python 选项。在名称为 Grammar Version 的下拉列表中选择 Python 的版本号，这里选择 2.7。在 Interpreter 下拉框中选择 Default 选项(如果没有下拉框，单击 Interpreter 下面的 Please configure an interpreter before proceeding 链接，在弹出一个 Configure interpreter 对话框后，单击 Quick Auto-Config)。选中 Create 'src' folder and add it to the PYTHONPATH 单选框，表示工程创建后会生成一个 src 目录，该目录即为 Python 的源代码目录，如图 9-15 所示。

最后单击 Finish 按钮，完成 Python 工程的创建。在 src 目录下新建 PyDev Module 模块，这里我们将其命名为 debug，在该文件中编辑代码，如图 9-16 所示。

2. 配置调试

在调试程序之前，需要设置 Python 解析器的路径，并导入 Python 环境变量下包含的库文件。执行 Window→Preferences 命令，弹出图 9-17 所示的 Preferences 窗口，在该窗口中可以对 Eclipse 的开发环境和各种插件进行配置，其中的节点 PyDev 就是 Python 插件的设置项。

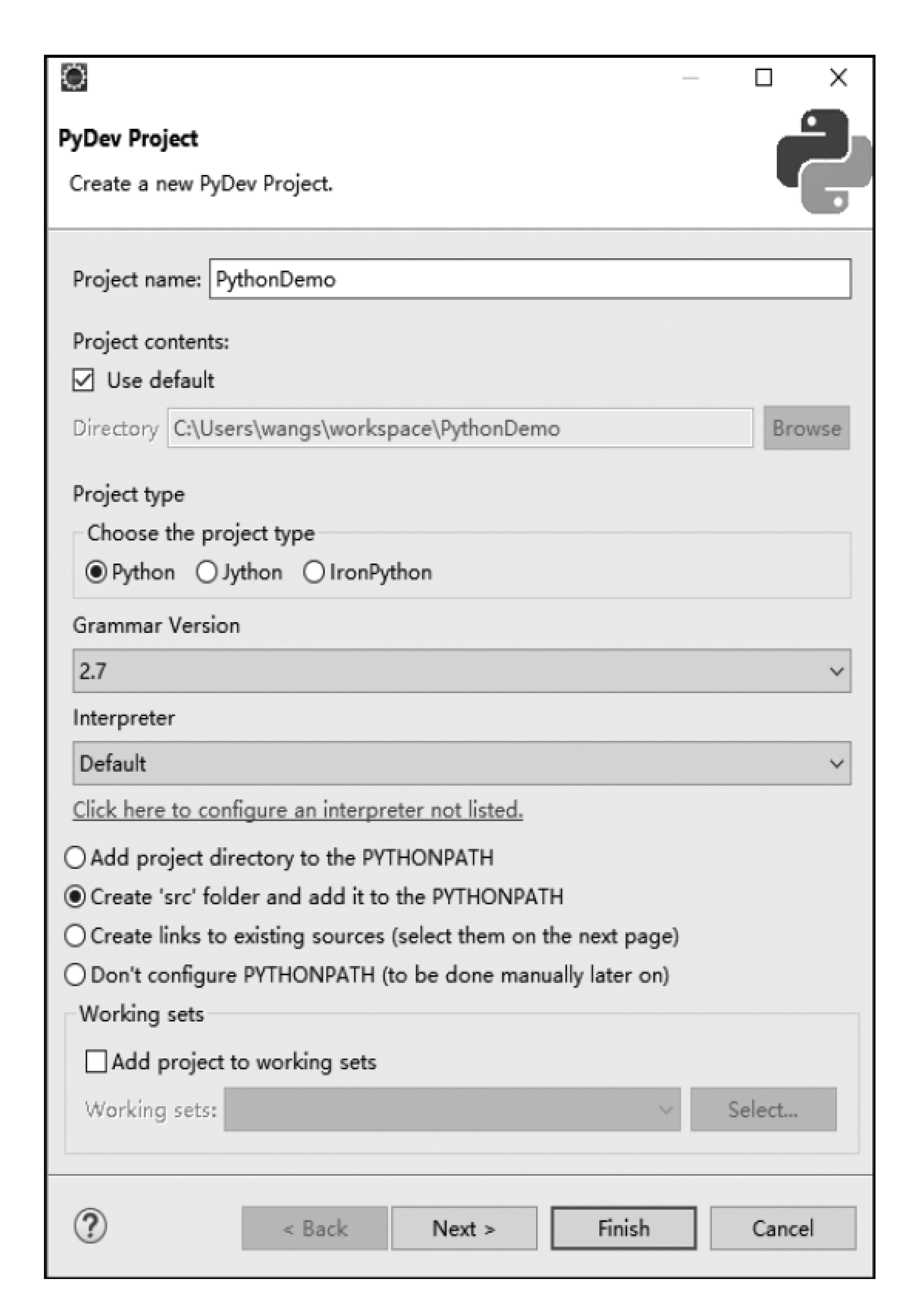


图 9-15　新建 PyDev Project

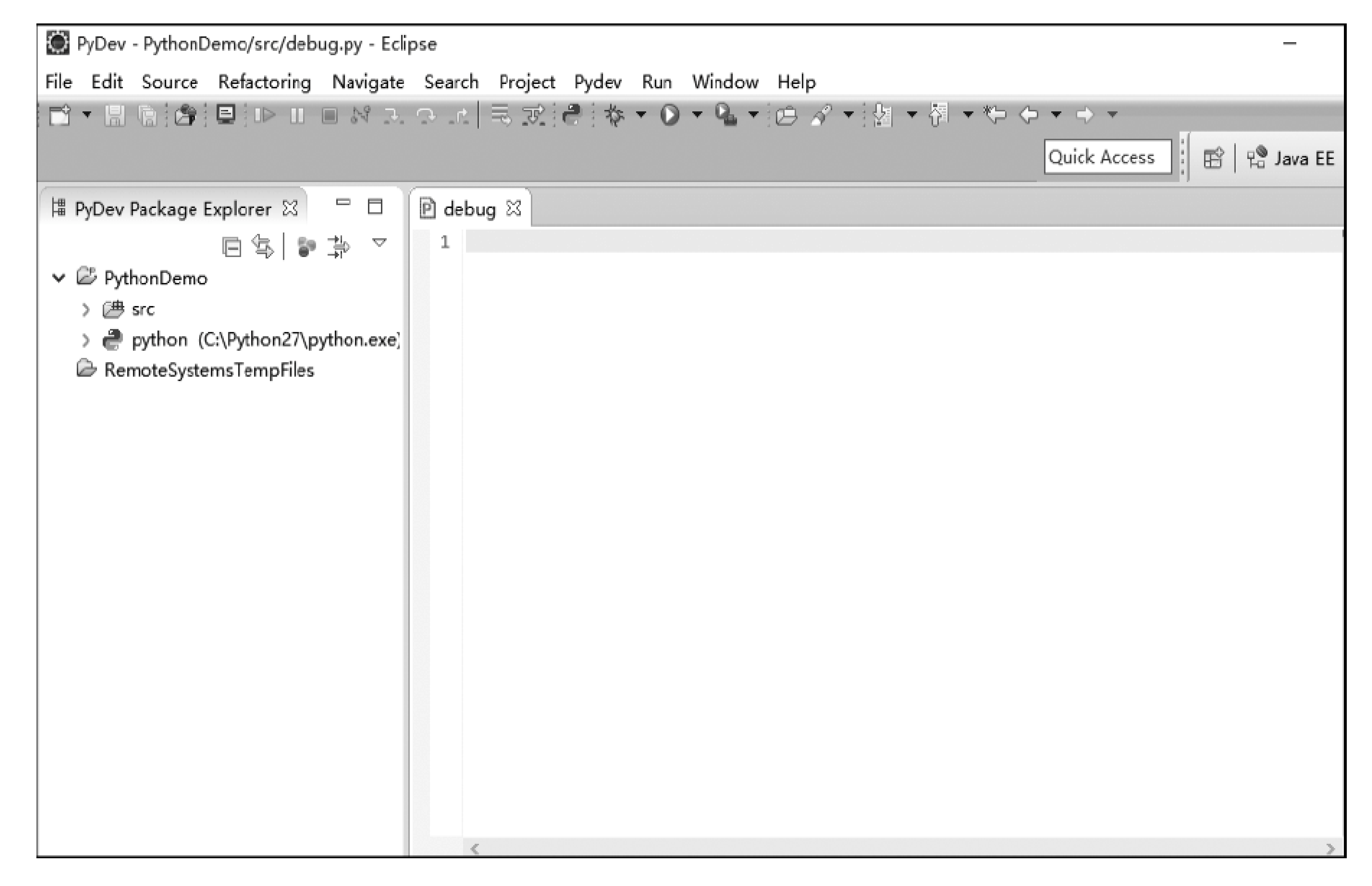
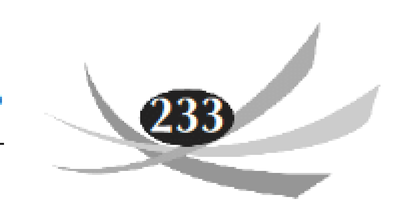


图 9-16　在 src 下创建 debug 模块文件



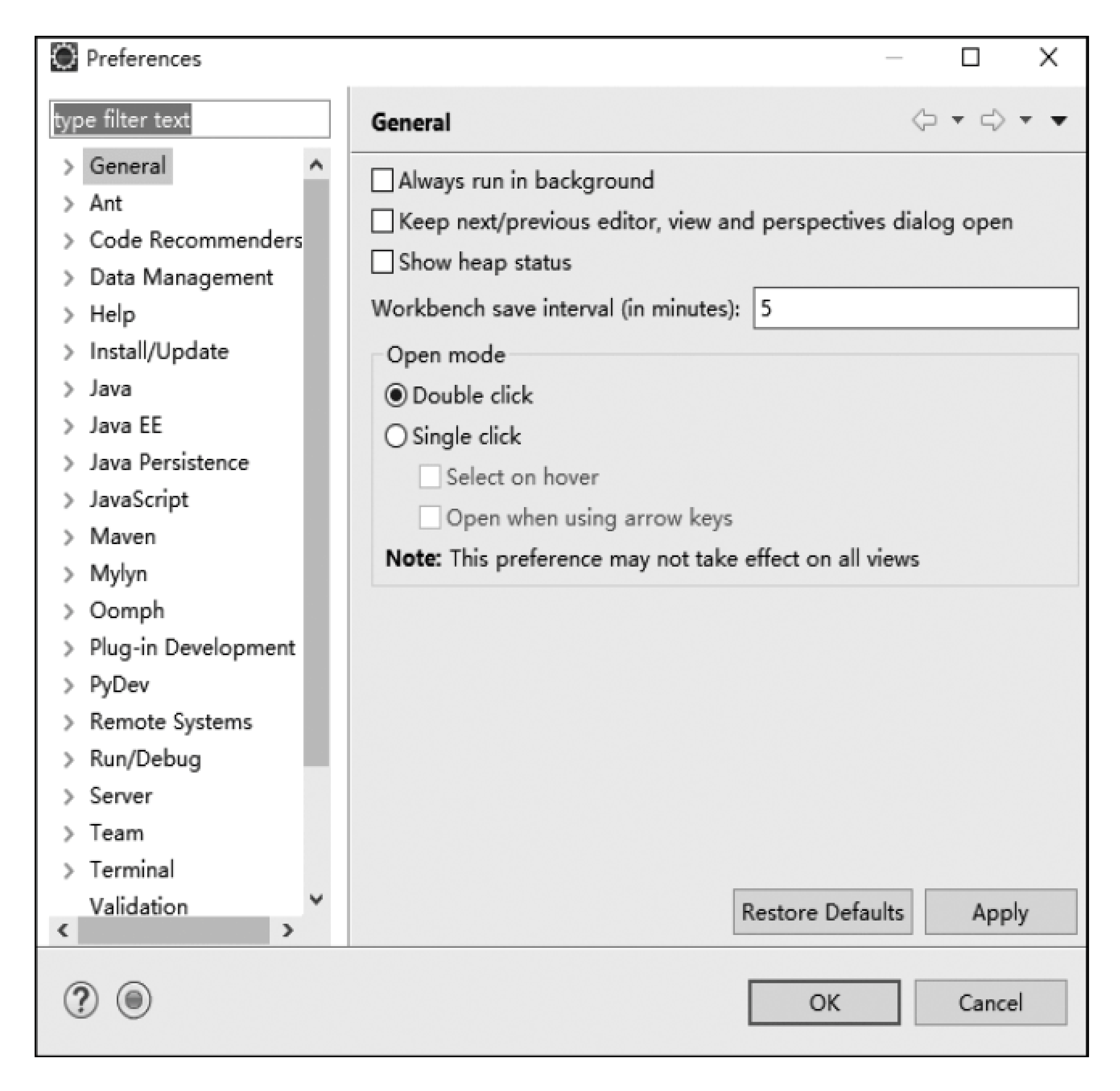


图 9-17　Preferences 窗口

展开节点 PyDev 后，选中 Interpreters 节点下的 Python Interpreter，然后单击 New...按钮，加入 python(默认已加入)和 pythonw.exe 所在的路径，然后单击 OK 按钮，如图 9-18 所示。

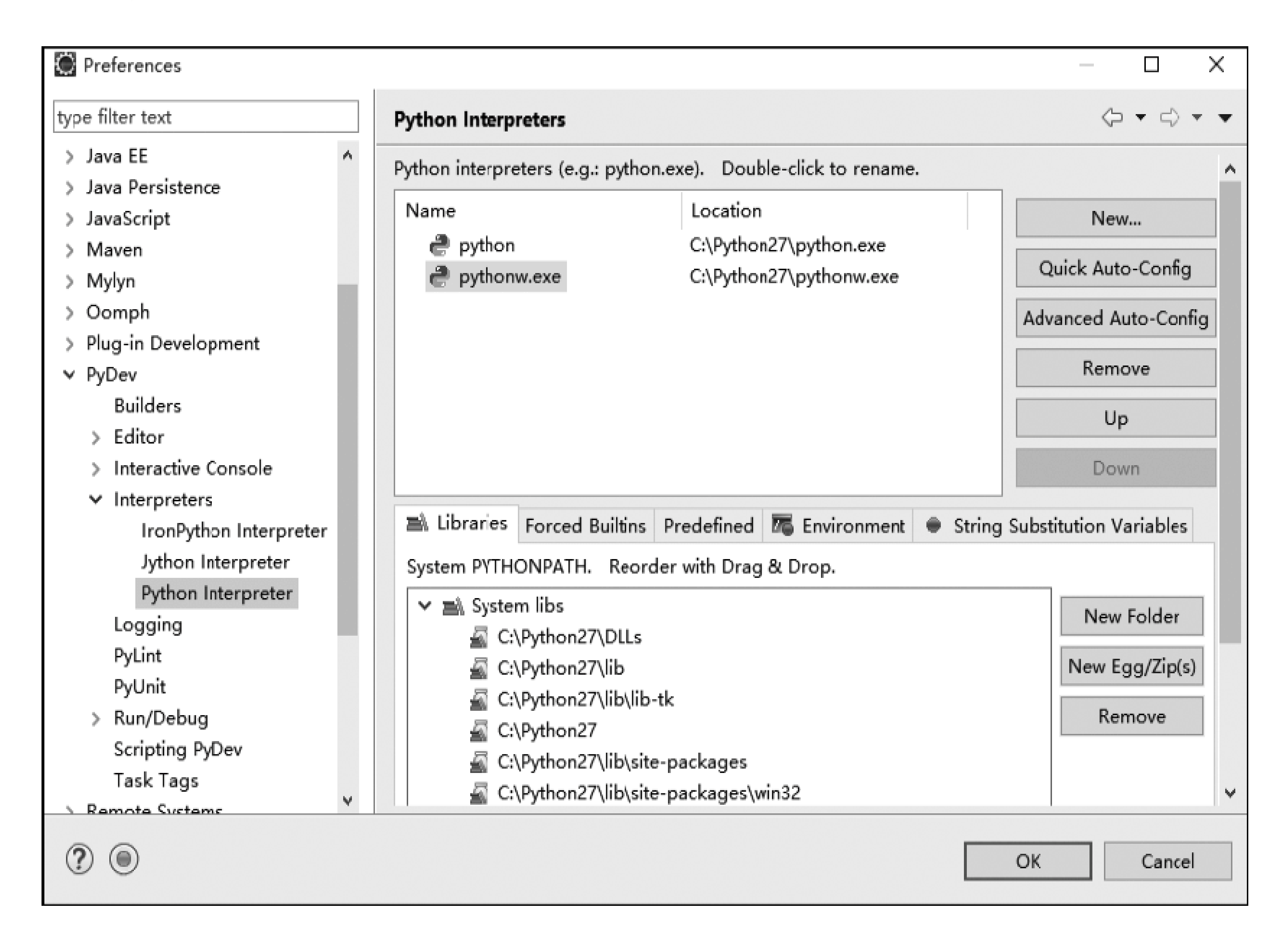


图 9-18　设置 Python 解析器

3. 调试程序

同样，我们调试例 9-6 的程序。把例 9-6 的程序代码复制到刚才创建的 debug.py 文

件中。然后设置断点(方法：在需要设置断点的代码行数左边的小灰色区域双击即可)。如图 9-19 所示。

```
1  '''
2  Created on 2016年1月17日
3
4  @author: wangsd
5  '''
6
7  class UserInfoError(Exception):
8      def __init__(self,code,message):
9          self.code = code
10         self.message = message
11
12     def __str__(self):
13         errorinfo = '{code:%d,message:%s}' % (self.code,self.message)
14         return errorinfo
15
16 def throwException():
17     try:
18         userInfoDict = {'admin':'admin'}
19         username = raw_input('please enter an username:')
20         password = raw_input('please enter a password:')
21         if  username!='admin':
22             raise UserInfoError,UserInfoError(40001,'username is not correct')
23         elif userInfoDict[username]!=password:
24             raise UserInfoError,UserInfoError(40002,'password is not correct')
25         else:
26             print 'login successfully'
27     except UserInfoError,e:
28         print 'There is a UserInfoError'
29         print e
30
31 throwException()
32
```

图 9-19　设置断点

在 Eclipse 工具栏中，单击 ![debug] 右侧的倒三角形，然后从打开的下拉菜单中选择 Python Demo dubug. py 选项，启动调试程序，如图 9-20 所示。

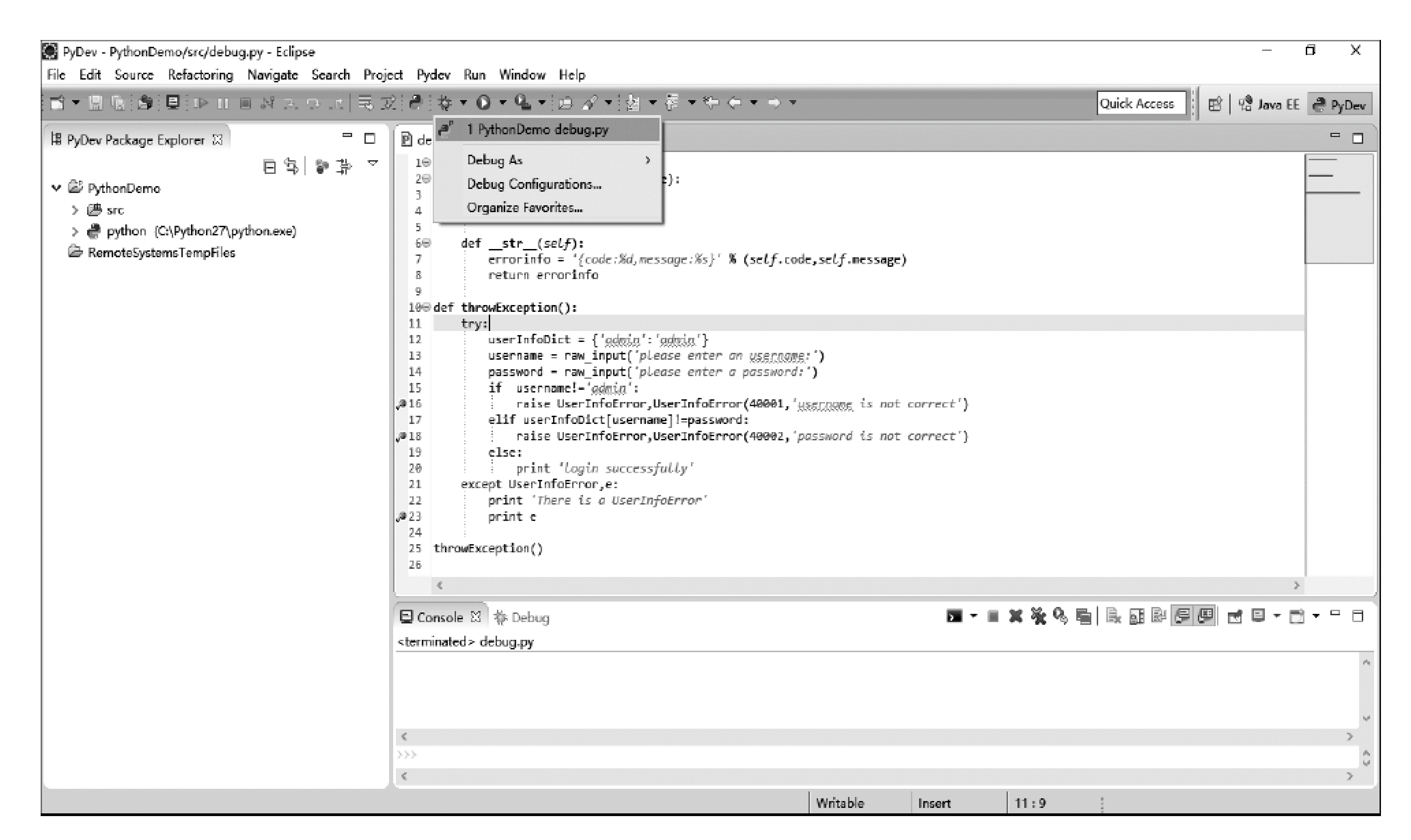


图 9-20　启动调试程序

当启动调试程序之后，系统在控制台(Console)中提示用户输入用户名和密码，按回车键后，Eclipse 提示是否进入调试模式，如图 9-21 所示。

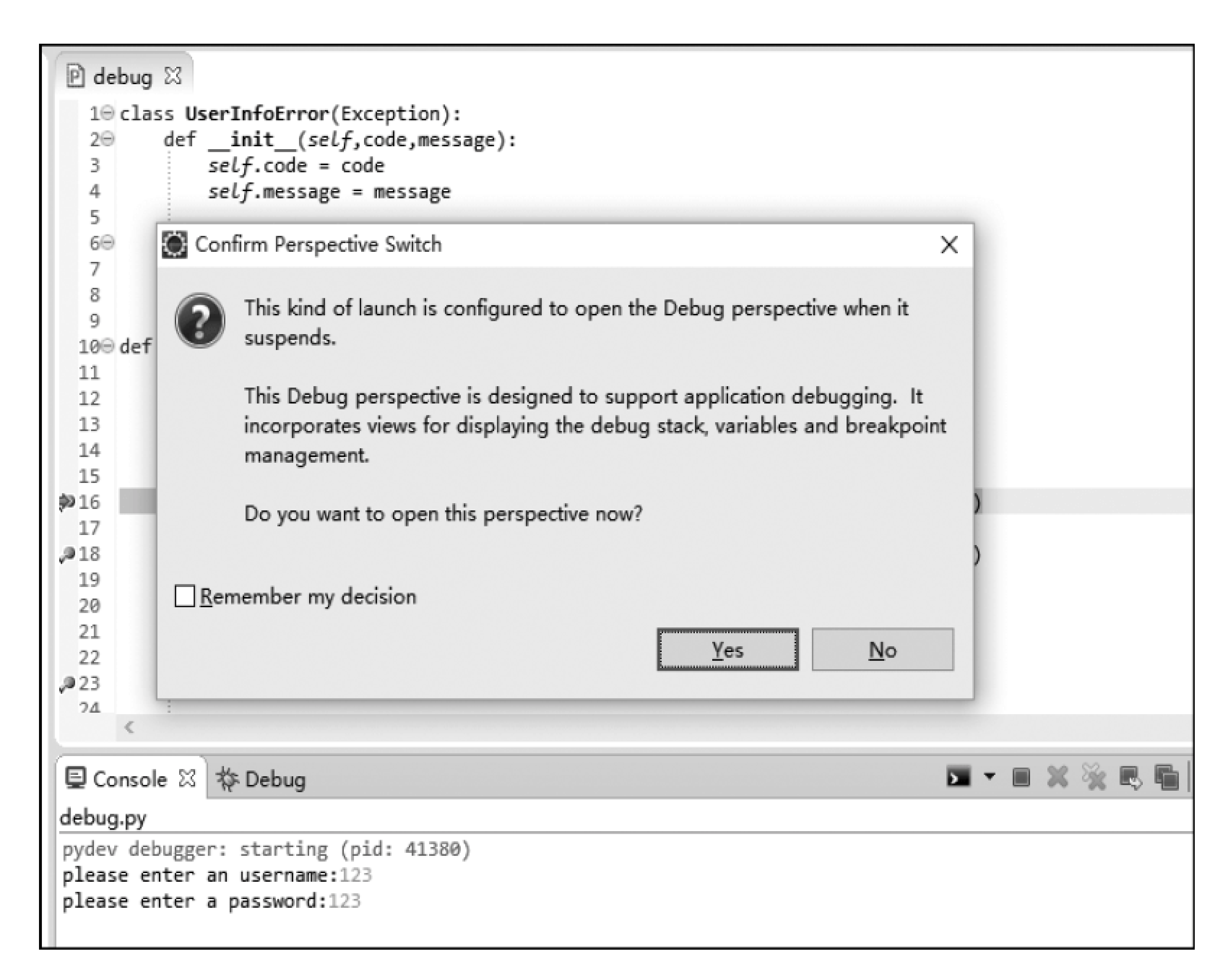


图 9-21　提示用户是否进入调试模式

单击 Confirm Perspective Switch 对话框中的 Yes 按钮，Eclipse 将自动切换到 Debug 窗口，如图 9-22 所示。其中 Breakpoints 标签页显示了当前程序中的断点信息；Variables 标签页显示了当前程序的变量值；Debug 标签页显示了 dubug. py 的主线程；Outline 标签页显示了当前程序中定义的函数名、类名等信息；Console 标签页显示了控制台的输出。

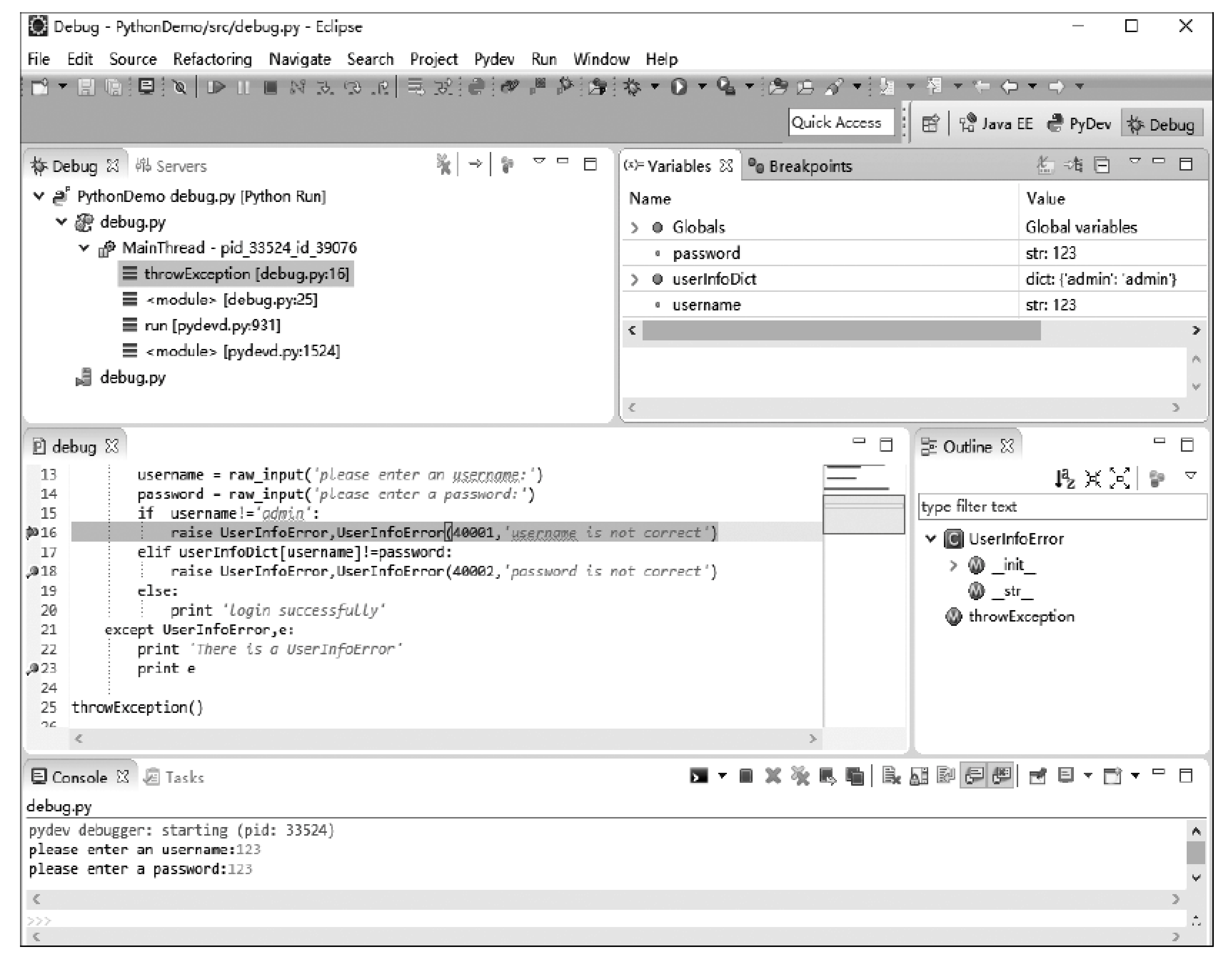


图 9-22　debub. py 的 Debug 调试窗口

可以看到此时程序运行到第一个断点，即第16行处就停止了。在Variables标签页看到username、password和userInfoDict等变量的值。

接着按F5键进行单步调试，可以看到程序的执行流程，并且在Variables标签页可以看到时时更新的变量及其对应的值，这在分析程序的业务逻辑是否正确时非常重要。当发现程序逻辑出现问题时，按Shift+F5键结束调试，修改程序后再次运行，验证程序是否正确。

9.5 本章小结

本章主要讲解了以下几个知识点。

(1) 异常。异常是程序运行过程中由于出现语法错误或逻辑错误而发生的事件，该事件可以中断程序指令的正常执行流程。标准异常类是系统提供的，在exceptions模块中定义，处于内建命令空间，可以直接使用。

(2) 异常处理。捕获异常可以通过三种语句捕获：try…except、try…except…else和try…except…finally语句。try…except语句把需要执行的代码放到try语句块中，而把出现异常后的代码放到except语句块中。当try语句块中的代码出现异常后，会执行except语句块中的代码，捕获异常并根据异常做出相应的处理。try…except…else、try…except…finally语句与try…except语句的执行过程很相似，区别在于else子句是在try语句块中没有发生异常时执行的；finally子句都会被执行，无论try语句块中是否发生异常。

(3) 抛出异常。当程序员在编写程序时希望在遇到错误的输入等原因时能够手动抛出异常。这时就可以通过raise语句抛出异常，给用户以友好的提示。raise语句更多地用于抛出用户自定义的异常类型。

(4) 自定义异常。当Python的内建异常类型不能满足我们的需要时，我们可以自定义异常的类型。自定义异常是一个类，它必须继承Exception或者BaseException类，按照命名规范，自定义异常通常以Error结尾，以显式地告诉程序员出现异常的类型，自定义异常只能通过raise语句手动抛出。

(5) 调试程序。程序调试就是在将编制的应用程序投入实际运行前，以手工编译程序等方式进行测试，修正语法错误和逻辑错误的过程。这是保证应用程序正确性必不可少的步骤。本章介绍了如何使用PythonWin和Eclipse for Python调试程序。

9.6 习　　题

一、解答题

1. 什么是异常？有哪些常用的标准异常类？并举例说明(至少5个)。
2. 捕获异常有哪几种方式？它们的区别是什么？
3. 如何手动抛出异常？
4. 如何声明自定义异常？

二、看程序写结果

1.

```
def testException():
    try:
        aint=123
        print aInt
        print aint
    except NameError,e:
        print 'There is a NameError'
    except KeyError,e:
        print 'There is a KeyError'
    except IndexError,e:
        print 'There is a IndexError'

testException()
```

print aInt 与 print aint 交换后结果又会如何?

2.

```
def tryExceptElseFinally():
    try:
        aList=[1]
        print aList[1]
        return 'try'
    except NameError,e:
        print 'There is a NameError'
        return 'NameError'
    except KeyError,e:
        print 'There is a KeyError'
        return 'KeyError'
    except IndexError,e:
        print 'There is a IndexError'
        return 'IndexError'
    except IOError,e:
        print 'There is a IOError'
        return 'IOError'
    else:
        print 'No exception occurred'
        return 'else'
    finally:
        print 'must be excecute'
        return 'finally'

print tryExceptElseFinally()
```

把 aList[1]改为 aList[0],结果又会如何?

3.

```
class BaseError(BaseException):
    def __init__(self,code,message):
        self.code=code
        self.message=message

    def __str__(self):
        errorinfo='{"code":"%d","message":"%s"}' % (self. code,self.message)
        return errorinfo

class InputError(BaseError):
    pass

class KeyNotFoundError(BaseError):
    pass

def throwException():
    try:
        phoneBookDict={'小李':15912345678,'小张':13812345 678,'小陈':18012345678}
        while 1:
            num=input('请输入数字选择功能(1:添加联系人,2:查询联系人,3:退出程序)')
            if num==1:
                name=raw_input('请输入联系人姓名:')
                phoneNum=input('请输入'+name+'的电话号码:')
                phoneBookDict[name]=phoneNum
                print '添加成功'
            elif num==2:
                name=raw_input('请输入联系人姓名:')
                if  name not in phoneBookDict.keys():
                    raise KeyNotFoundError,KeyNotFoundError (40001,'电话簿没有'+name+
                   '的信息')
                else:
                    print phoneBookDict[name]
                    break
            elif num==3:
                print '退出程序'
                break
            else:
                raise InputError,InputError(40004,'输入有误')
    except (KeyNotFoundError,InputError),e:
        print e

throwException()
```

（1）输入 4 时，输出的结果的什么？

（2）依次输入 2、小李，输出的结果是什么？

（3）依次输入 2、小何，输出的结果是什么？

三、上机练习

为第二题的第 3 小题增加删除联系人功能。如果输入的联系人不在 phoneBook Dict 字典中，先输出删除失败，然后再手动抛出一个 KeyNotFoundError；如果输入的联系人在 phoneBookDict 字典中，则将其信息删除，并输出删除功能，然后退出程序。如图 9-23 所示。

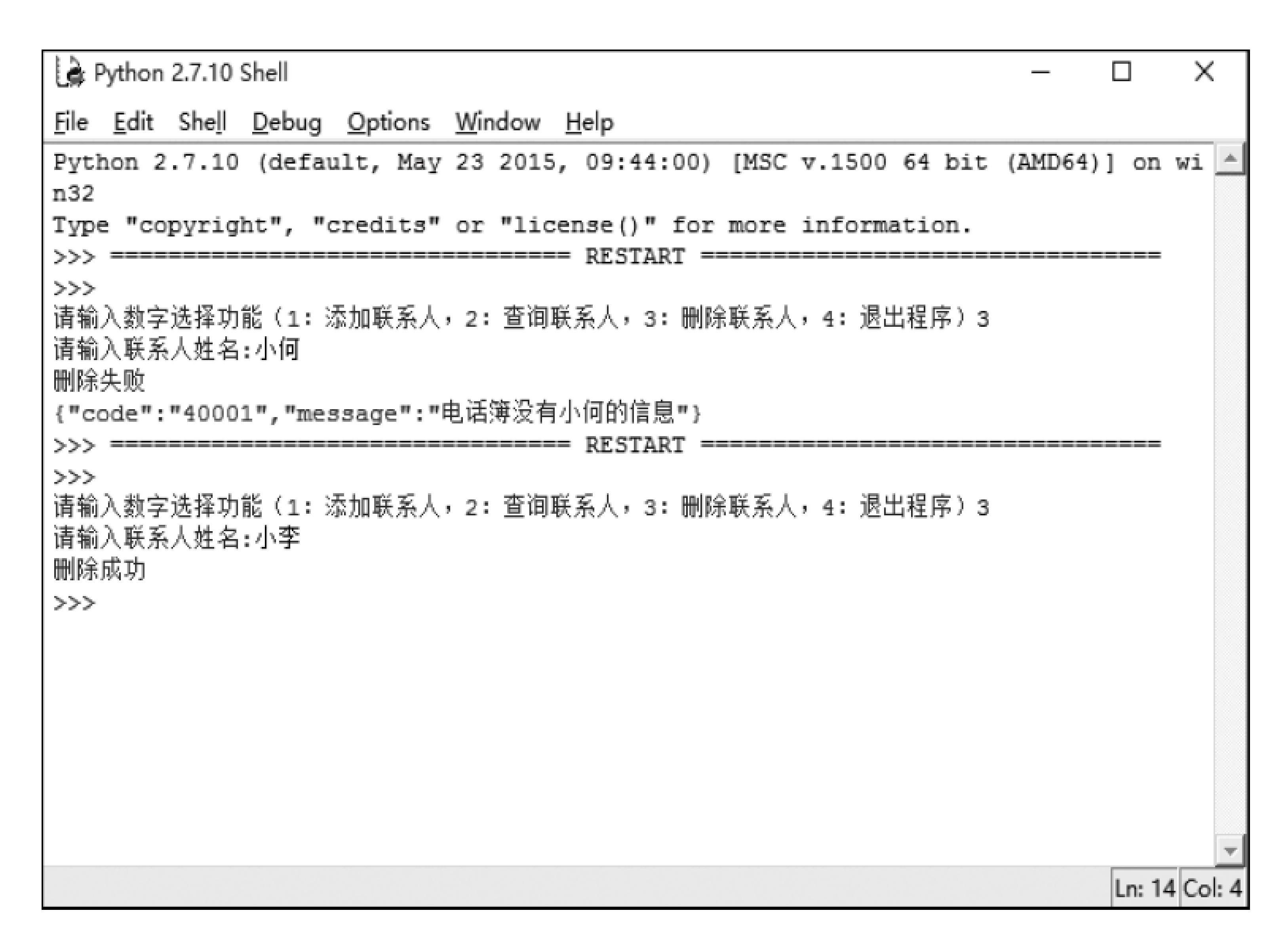


图 9-23　上机练习题的示意图

第 10 章 文　件

本章学习目标

- 理解文件的概念
- 掌握文件的打开和关闭
- 掌握文件的读写
- 掌握文件的备份和删除
- 掌握文件夹的创建和删除

在前面的章节中所用到的输入和输出都是以终端作为对象的，即从终端键盘输入数据，运行结果输出到终端上。本章将介绍程序设计中一个重要的概念——文件，它是一个非常常用的、用于存储数据的媒介。在实际的应用程序开发过程中经常会涉及对文件的操作，因此，本章首先介绍文件的基本概念，然后重点介绍对文件的操作，包括文件的打开与关闭，文件的读写等内容。

10.1 文件概述

文件是程序设计中一个重要的概念。所谓“文件”一般指存储在外部介质上的数据的集合。一批数据是以文件的形式存放在外部介质（如磁盘）上的。操作系统是以文件为单位对数据进行管理的，也就是说，如果想找存在外部介质上的数据，必须先按文件名找到所指定的文件，然后再从该文件中读取数据。要向外部介质上存储数据也必须先建立一个以文件名作为标识的文件，才能向它输出数据。

前面的章节中所用到的输入和输出都是以终端作为对象的，即从终端键盘输入数据，运行结果输出到终端上。从操作系统的角度看，每一个与主机相联的输入输出设备都看作是一个文件。例如，终端键盘是一个输入文件，显示屏和打印机又是不同的输出文件。

在程序运行时，常常需要将一些中间数据或最终的结果输出到磁盘上存放起来，以后需要时再从磁盘中输入到计算机内存。这就要用到磁盘文件。

文件可以看作是一个字符（字节）的序列，即由一个个字符（字节）的数据顺序组成。根据数据的组织形式，可分为 ASCII 文件和二进制文件。ASCII 文件又称为文本文件，它的每一个字节放一个 ASCII 代码，代表一个字符。二进制文件是把内存中的数据按其在内存中的存储形式原样输出到磁盘上存放。

10.2　文件的打开与关闭

和其他高级语言一样，对文件读写之前应该“打开”文件，在使用结束之后应“关闭”该文件。

10.2.1　文件的打开

在实际的应用程序开发过程中，凡是提到文件操作，必然涉及文件的打开和创建。在Python中，可以用open和file内建函数打开文件，它们具有相同的功能，可以任意替换。所以，这里以open函数为例来介绍文件的打开。

一般情况下，使用open函数时只需传入文件名参数，而无须添加其他任何参数，就可以获取文件的内容。但是，如果要向文件写入内容，就必须指定一个访问模式参数，用来声明将对文件进行什么样的操作。该函数会返回一个指定的文件对象，open函数的使用语法如下：

```
open(filename, accessmode='r', buffering=-1)
```

其中，filename参数表示需要打开的文件名称；accessmode是一个可选的参数，表示打开的模式，其值是一个字符串，默认值是'r'，即只读模式，所有打开的模式如表10-1所示；buffering也是一个可选的参数，用于指示方位文件所采用的缓冲方式，0表示不缓冲，1表示只缓冲一行数据，任何其他大于1的值表示使用给定值作为缓冲区大小。给定负值代表使用系统默认缓冲机制，该参数的默认值为－1。

表10-1　文件打开模式

文件模式	说　　明
r	以读方式打开一个文本文件
rU	以读方式打开一个文本文件，并且支持文件内容含特殊字符（如换行符）
w	以写方式打开一个文本文件
a	以追加方式打开一个文本文件
r＋	以读写方式打开一个文本文件
w＋	以读写方式新建一个文本文件
a＋	以读写方式打开一个文本文件
rb	以读方式打开一个二进制文件
wb	以写方式打开一个二进制文件
ab	以追加方式打开一个二进制文件
rb＋	以读写方式打开一个二进制文件
wb＋	以读写方式新建一个二进制文件
ab＋	以读写方式打开一个二进制文件

说明：

（1）用“r”方式打开的文件只能读取其中的数据，而不能用作向该文件输出数据，而且该文件应该已经存在，不能用“r”方式打开一个并不存在的文件，否则会抛IOError异常，提示该文件不存在。

（2）用“w”方式打开的文件只能用作向该文件输出数据，而不能读取其中的数据。如果原来不存在该文件，则在打开时新建一个以指定的名字命名的文件。如果原来已存在一个以该文件名命名的文件，则在打开时将该文件删除，然后重新建立一个新文件。

（3）如果希望向文件末尾添加新的数据，即不删除原有的数据，则应该用“a”打开。但此时该文件必须已存在，否则同样会抛IOError异常。打开时，位置指针移到文件末尾。

（4）用“r＋”、“w＋”和“a＋”方式打开的文件既可以读取其中的数据，也可以向其写入数据。用“r＋”方式时该文件应该已经存在，以便能够读取其中的数据。用“w＋”方式则新建一个文件，先向此文件写入数据，然后可以读取此文件中的数据。用“a＋”方式打开的文件，原来的文件不被删除，位置指针移到文件末尾，可以添加，也可以读取。

（5）如果不能实现“打开”的任务，open函数将会抛出一个IOError异常。出错原因可能是用“r＋”方式打开一个并不存在的文件；磁盘出故障；磁盘已满无法建立新文件等。所以，应该把打开文件的代码放到try语句块中，使用except子句捕获IOError异常，并在except子句中用print语句在终端上输出“cannot open this file”，最后执行exit函数关闭所有文件，并终止正在执行的程序，待用户检查出错误，修改后再运行。

（6）在读取文件中的数据时将回车换行符转换为一个换行符，在向文件输出数据时把换行符转换成回车和换行两个字符。在用二进制文件时，不进行这种转换，在内存中的数据形式与输出到外部文件中的数据形式完全一致，一一对应。

（7）在程序开始运行时系统自动打开三个标准文件：标准输入(stdin)、标准输出(stdout)、标准错误输出(stderr)。通常这三个文件都与终端相联系。因此以前我们所用到的从终端输入或输出时都不需要打开终端文件。如果程序中指定要从stdin文件输入数据，就是指从终端键盘输入数据。

下面通过一个例子来说明open函数的用法。

```
#coding:utf-8
#例 10-1  open 函数
def testOpen():
    try:
        f1=open('D:\\a.txt')
        f2=open('D:\\b.txt','w')
        f3=open('D:\\c.txt','a+')
        #对文件操作
    except IOError,e:
        print e
        exit()
    finally:
        #关闭文件
```

```
        f1.close()
        f2.close()
        f3.close()

testOpen()
```

程序第 4 行以默认的“r”方式打开 D 盘下的 a.txt 文件，如果该文件不存在，则会抛出 IOError 异常。以这种方式打开的文件，只能对其进行读取操作，否则也会抛出 IOError 异常。第 5 行以“w”方式打开 D 盘下的 b.txt 文件，如果该文件不存在，则会创建以 b.txt 命名的文件。以这种方式打开的文件，只能对其进行写入操作，否则会抛出 IOError 异常。第 6 行以“a+”方式打开 D 盘下的 c.txt 文件，如果该文件不存在，则会创建以 c.txt 命名的文件。以这种方式打开的文件，既可以对其进行读取，也可以写入(在文件末尾写入)。文件的打开操作及其读写操作都放在 try 语句块中，然后在 except 子句捕获异常，最后在 finally 子句中关闭所打开的文件。

10.2.2　文件的关闭

在使用完一个文件后应该关闭它，以防止它再次被误用。“关闭”就是使指向该文件对象的引用不再指向该文件，也就是文件引用变量与文件对象“脱钩”，以后不能再通过该引用对原来与其相联系的文件进行读写操作，除非再次打开，使该文件引用变量重新指向该文件。

Python 提供了 close 函数关闭文件。close 函数调用的一般形式为：

```
fileRef.close()
```

其中，fileRef 是指向所打开的文件的引用变量。

例如：

```
f=open('D:\\c.txt','r+')                    #假设 c 文件存在
f.close()
```

把通过 open 函数返回的文件对象赋给变量 f，使 f 指向所打开的文件对象，然后就可以对文件进行操作，最后执行 f 的 close 函数，关闭该文件，即变量 f 不再指向该文件。

应该养成在程序终止之前关闭所有文件的习惯，如果不关闭文件将会丢失数据。因为在向文件写数据时，是先将数据输出到缓冲区，待缓冲区充满后才正式输出该文件。如果当数据未充满缓冲区而程序结束运行，就会将缓冲区中的数据弄丢。用 close 函数关闭文件，可以避免这个问题，它先把缓冲区中的数据输出到磁盘文件，然后才使该变量不再指向所指定的文件。

10.3　文件的读写

文件打开之后，就可以对它进行读写了。常用的读写函数如下所述。

10.3.1　文件的读取

将文件的内容读入到计算机内存有三个函数，分别是 read()、readline()和 readlines

()函数,但它们对文件的读取方式各不相同。其中,read()函数可一次性读取数据,readline()函数按行读取数据,而 readlines()函数则以多行的方式一次性读取数据。下面将分别介绍这三个函数。

1. read()函数

read()函数可以一次性将文件中的所有数据读取出来(前提是位置指针指向该文件内容的起始处,关于位置指针的内容将在 10.4 节介绍),这是最简单的文件读取方式。该函数的一般调用格式如下:

```
content=fileRef.read([size])
```

其中,size 参数表示读取该文件中的前几个字节的数据,该参数是一个可选参数,不指定(默认值为-1)或指定负值,将读取文件的所有内容。

下面通过一个例子来说明 read 函数的用法。

```
#coding:utf-8
#例 10-2  read 函数
def testRead():
    try:
        f=open('D:\\a.txt','r')
        content=f.read()
        print '未指定 read 函数的 size 参数:',content
        f.seek(0)        #该行的作用是把位置指针移回到文件内容的起始处
        conOneByte=f.read(5)
        print '指定 read 函数的 size 参数为 5 字节:',conOneByte

    except IOError,e:
        print e
    finally:
        f.close()

testRead()
```

假设 D 盘下 a.txt 文件的内容第一行 abc(有回车换行符\r\n),第二行 def(无回车换行符),则输出结果为:

```
未指定 read 函数的 size 参数:
abc
def
指定 read 函数的 size 参数为 5 字节:
abc
d
```

当不指定要读取的字节数时,且此时位置指针指向该文件内容起始处,默认读取所有的内容,所以输出的内容和文件内容一致。注意,此时位置指针移到文件内容最后一个字符(f)的后面,需要使用 seek()函数把位置指针移到文件内容的起始处,否则读取不到文

件的内容。然后指定读取的字节数为 5,因为该文件是文本文件,一个字节对应一个字符,且读取文本文件会把回车换行符(\r\n)转换为一个换行符(\n),所以输出的结果,第一行 abc(包括\n 换行符),第二行 d,共 5 个字符。

2. readline()函数

readline()函数也可以读取文件的内容,但它的读取方式不同于 read()函数,它每次只读取文件中的一行数据,该函数的一般调用格式如下:

```
content=fileRef.readline([size])
```

这里的 size 参数也是一个可选参数,但与 read()函数中的 size 参数有些不同,它表示读取当前位置指针指向的行的前几个字节的数据,不指定(默认值为-1)或指定负值,将读取当前位置指针指向的行的所有内容。把例 10-2 的 read()函数改为 readline()函数,并把 f.seek(0)行代码注释掉,程序运行结果如下:

```
未指定 readline 函数的 size 参数:
abc

指定 readline 函数的 size 参数为 5 字节:
def
```

当刚打开该文件时,位置指针指向第一行开头。不指定读取当前行的前几个字符,默认就读取当前行的所有字符,包含换行符,所以输出结果为 abc,并有换行效果。此时位置指针指向第二行开头,当指定读取当前位置指针指向的行的前 5 个字节的数据时,而此行只有 3 个字符,将其全部输出,结果就是 def。

3. readlines()函数

该函数和前面介绍的两个文件读取函数又有些不同,它是一次性读取当前位置指针指向处后面的所有内容,函数返回的是一个由每行数据组成的一个列表。通常使用迭代的方式读取其中的内容。该函数的一般调用格式如下:

```
listContent=fileRef.readlines()
```

该函数没有参数。

把例 10-2 的程序修改成如下所示:

```
#coding:utf-8
#例 10-3  readlines 函数
def testReadlines():
    try:
        f=open('D:\\a.txt','r')
        #读取第一行的内容
        f.readline()
        listContent=f.readlines()
        for oneLine in listContent:
            print oneLine
```

```
    except IOError,e:
        print e
    finally:
        f.close()

testReadlines()
```

假设此时 D 盘下 a.txt 文件前 5 行的内容分别是 abc、def、ghi、jkl、mno，程序先使用 readline()函数读取第一行的内容，此时位置指针指向第二行的开头，然后使用 readlines()函数把剩下的内容全部读取出来，并赋给 listContent 变量，此时 listContent 变量的每个元素都有 4 个字符(含最后的换行符)，再通过 for 循环把它们输出(有换行效果)，其输出结果如下：

```
def

ghi

jkl

mno
```

上面介绍的都是读取文本文件，那么这些函数又是如何读取二进制文件的呢？下面通过一个例子说明。假设 D 盘下有 a.PNG 的图片文件(二进制文件中的其中一个)。

```
#coding:utf-8
#例 10-4  读取二进制文件
def testReadBinaryFile():
try:
        #打开文件方式为'rb'(读取二进制文件)
        f=open('D:\\a.PNG','rb')
        index=0
        print '-----------以下是 a.PNG 图片文件的数据(十六进制格式)----------'
        while True:
            #每次读取一个字节
            temp=f.read(1)
            if len(temp)==0:
                break
            else:
                #将读取的数据转换为十六进制的数据
                print "%3s"%temp.encode('hex'),
                index=index+1
            #控制每行输出 16 组数据
            if index==16:
                index=0
                print
    except IOError,e:
```

```
        print e
    finally:
        f.close()

testReadBinaryFile()
```

程序的运行结果如图 10-1 所示(部分数据)。

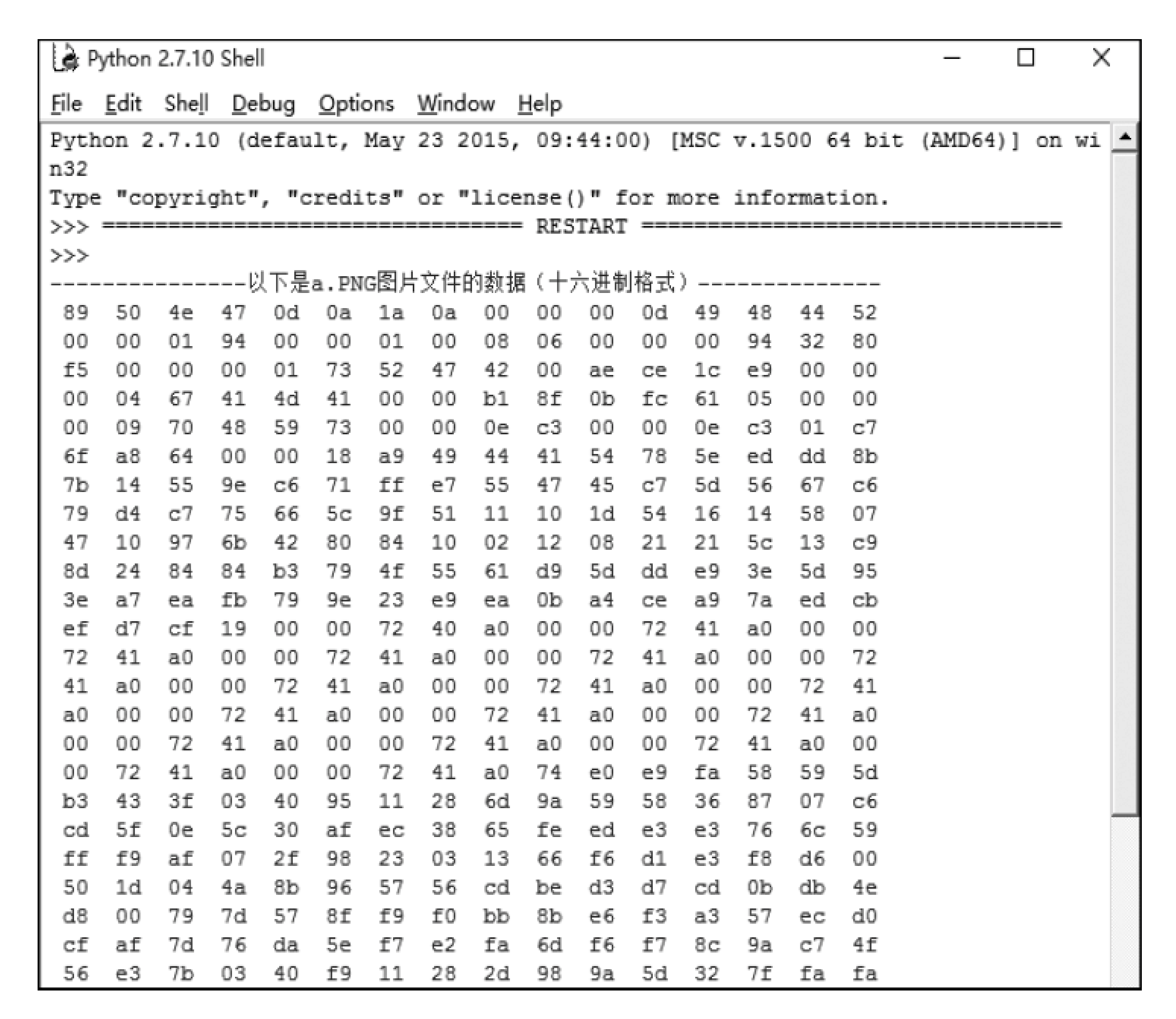


图 10-1　读取二进制文件输出的结果

10.3.2　文件的写入

将内容写入到文件中有两个函数,分别是 write()和 writelines()函数,这两个函数的区别在于操作的对象不同,write()函数是把一个字符串写入到文件中,而 writelines()函数则是把列表中的字符串内容写入到文件中。注意这里并没有 writeline 函数,因为它等价于 write 函数,把以换行符结束的单行字符串写入文件。下面将分别介绍 write()函数和 writelines()函数。

1. write()函数

write()函数是把一个字符串写入到文件中。在使用该函数前,open 函数不能以'r'的方式打开一个文件。该函数的一般调用格式如下:

```
fileRef.write(content)
```

其中,content 参数表示要写入的内容,可以是一个字符串或指向字符串对象的变量,甚至还可以是它们组成的合法的字符串表达式。

下面通过一个例子来说明 write 函数的用法。

```
#coding:utf-8
```

```
#例 10-5  write函数
def testWrite():
    try:
        f=open('D:\\b.txt','w+')
        f.write('Python is great!')
        f.write('I agree!')
        f.write('\n')
        #通过 write函数模拟 writeline函数
        f.write('So,I will try my best to learn Python well!\n')
        print '写入成功'
        #读取其内容并输出
        f.seek(0)
        content=f.read()
        print '写入内容如下:'
        print content
    except IOError,e:
        print e
    finally:
        f.close()

testWrite()
```

程序第一次调用 write 函数把 Python is great! 写入到 b.txt 文件中,此时位置指针指向感叹号(!)的后面,所以第二次调用 write 函数时把 I agree! 写到感叹号的后面,第三次写入一个换行符,此时位置指针指向第二行开头,第四次调用时,直接把换行符加到所要写入的字符串后面,模拟 writeline 函数。然后调用 seek 函数,把位置指针重新指向文件内容的起始处,再通过 read 函数读取前面写入的所有内容,并将其输出,输出结果如下:

```
写入成功
写入内容如下:
Python is great!I agree!
So,I will try my best to learn Python well!
```

2. writelines()函数

writelines()函数也可以用于对文件进行写入操作,与 write 函数不同的是,该函数是把一个列表的内容都写入到文件中。该函数的一般调用格式如下:

```
fileRef.writelines(strList)
```

该函数接受一个字符串列表作为参数,将它们写入到文件中。

下面通过一个例子说明 writelines 函数的用法:

```
#coding:utf-8
#例 10-6  writelines函数
```

```
def testWritelines():
    try:
        f=open('D:\\b.txt','w+')
        strList=['Python is great!\n','I agree!\n','So,I will try my best to learn Python!
']
        f.writelines(strList)
        print '写入成功'
        #读取其内容并输出
        f.seek(0)
        content=f.read()
        print '写入内容如下:'
        print content
    except IOError,e:
        print e
    finally:
        f.close()

testWritelines()
```

该程序定义了一个含三个字符串元素的列表,并且前两个字符串都以换行符结束,这样才能以分行的形式存储它们。然后调用 seek 函数,把位置指针重新指向文件内容的起始处,再通过 read 函数读取前面写入的所有内容,并将其输出,输出结果如下:

```
写入成功
写入内容如下:
Python is great!
I agree!
So,I will try my best to learn Python!
```

10.4　文件的定位

文件中有一个位置指针,指向当前读写的位置。如果顺序读写一个文件,每次读写一个字符(字节),则读写完一个字符(字节)后,该位置指针自动移动指向下一个字符(字节)。除了这种顺序读写的方式,Python 提供了 seek 函数以实现随机读写文件的功能。此外,还提供了 tell 函数,通过这个函数可以得知位置指针的当前位置,下面分别介绍这两个函数。

10.4.1　seek 函数

对文件可以进行顺序读写,也可以进行随机读写,关键在于控制文件的位置指针。如果位置指针是按字节位置顺序移动的,就是顺序读写;如果能将位置指针按需要移动到任意位置,就可以实现随机读写。所谓随机读写,是指读写完上一个字符(字节)后,并不一定要读写其后续的字符(字节),而可以读写文件中任意位置上所需要的字符(字节)。

使用 seek 函数可以实现改变文件的位置指针,该函数在前面的例子中有用到。

seek 函数的调用形式为：

```
fileRef.seek(offset, startpoint=0)
```

其中，startpoint(起始点)用 0、1 或 2 代替，0 表示文件开始，1 表示当前位置，2 表示文件末尾，默认值为 0，即文件开始。offset(偏移量)指以 startpoint 为基点，往后移动的字节数。

seek 函数多用于二进制文件，因为文本文件要发生字符转换(回车换行符与换行符相转换)，计算位置时容易发生混乱。

下面通过一个例子说明 seek 函数的用法。

```
#coding:utf-8
#例 10-7  seek 函数
def testSeek():
    try:
        f=open('D:\\b.txt','w+')
        strList=['abc\n','def\n','ghi']
        f.writelines(strList)
        #使位置指针移到文件开头
        f.seek(0)
        content1=f.read(1)
        print 'content1:',content1
        #使位置指针移到当前位置后一个字符处
        f.seek(1,1)
        content2=f.read(1)
        print 'content2:',content2
        #使位置指针移到文件末尾前一个字符处
        f.seek(-1,2)
        content3=f.read(1)
        print 'content3:',content3
    except IOError,e:
        print e
    finally:
        f.close()

testSeek()
```

程序第一次调用 seek 函数，只传一个值 0，表示把位置指针移到以文件开头(默认值 0)为基点，往后偏移量为 0 个字节的位置，即文件开头的位置，然后读取一个字节的数据并赋给 content1，因此 content1 的内容就是 a。第二次调用 seek 函数，把位置指针移到以当前位置(字符 a 的后面，b 的前面)为基点，往后偏移量为 1 个字节的位置，即字符 b 的后面位置，然后读取一个字节的数据并赋给 content2，因此 content2 的内容就是 c。第三次调用 seek 函数，把位置指针移到以文件末尾为基点，往后偏移量为－1 个字节的位置，即文件末尾前一个字符的位置，然后读取一个字节的数据并赋给 content3，因此 content3

的内容就是 i。程序运行结果如下：

```
content1: a
content2: c
content3: i
```

10.4.2　tell 函数

tell 函数的作用是得到位置指针的当前位置，用相对于文件开头的位移量（单位是字节）来表示。由于文件中的位置指针经常移动，人们往往不容易知道其当前位置。用 tell 函数可以得到当前位置。

tell 函数的调用形式为：

```
fileRef.tell()
```

下面通过一个例子说明 tell 函数的用法。

```
#coding:utf-8
#例 10-8  tell 函数
def testTell():
    try:
        f=open('D:\\b.txt','w+')
        strList=['abc\n','def\n','ghi']
        f.writelines(strList)
        print '位置指针相对于文件开头的偏移量(单位:字节):',f.tell()
        f.seek(3)
        print '位置指针相对于文件开头的偏移量(单位:字节):',f.tell()
        content=f.read()
        print content
        f.seek(4)
        print '位置指针相对于文件开头的偏移量(单位:字节):',f.tell()
        content=f.read()
        print content
    except IOError,e:
        print e
    finally:
        f.close()

testTell()
```

程序把列表 strList 的内容写入文件中，在写入时，把一个换行符转换成回车换行两个字符，有 strList 列表有两个换行符，转换后就有 4 个字符，再加上普通字符（9 个），共 13 个字符。执行完写入操作时，位置指针就移到文件末尾，因此第一次调用 tell 函数，返回 13。然后调用 seek 函数，使位置指针移到文件开头后的 3 个字符处，即字符 c 的后面，回车符的前面。接着调用 tell 函数，返回就是 3，读取并输出剩下的内容（两个回车换行

符转换为换行符）。再一次调用 seek 函数，使位置指针移到文件开头后的 4 个字符处，即回车符的后面，换行符的前面。同样调用 tell 函数，返回就是 4，最后读取剩下的内容（第一个回车符已读取不到，只把第二个回车换行符转换为换行符）。虽然两次读取文件剩下的内容是不同的（相差一个回车符），但输出结果都是一样的，程序运行结果如下：

```
位置指针相对于文件开头的偏移量(单位:字节): 13
位置指针相对于文件开头的偏移量(单位:字节): 3

def
ghi
位置指针相对于文件开头的偏移量(单位:字节): 4

def
ghi
```

10.5 文件的备份和删除

文件备份（复制）和删除在我们的日常生活中是两个很常用的文件操作功能，下面将分别介绍在 Python 中如何通过程序来实现文件的备份和删除功能。

10.5.1 文件的备份

文件备份是指为防止系统出现操作失误或系统故障导致文件丢失，而将全部或部分文件集合从应用主机的硬盘或阵列复制到其他的存储介质的过程。很多企业计算机里面重要的文件、文档或历史记录，对本企业至关重要的，一时不慎丢失，都会造成不可估量的损失，轻则辛苦积累起来的心血付之东流，严重的会影响企业的正常运作，给工作造成巨大的损失。为了保障生产、销售、开发的正常运行，企业数据备份相当重要。

文件备份这么重要，那么在 Python 程序中是如何实现的呢？实际上，文件备份完全可以通过前面介绍的文件读写操作来实现。下面用一个例子来说明如何通过文件的读写操作实现文件的备份功能。为了和接下来介绍的 copyfile 函数进行对比，我们把 D 盘下文件大小为 456MB 的 a.mp4 文件备份到 E 盘下，同时输出备份所需的时间。

```
#coding:utf-8
#例 10-9  通过 read 和 write 函数实现文件备份
import time                                         #导入 time 模块
def testFileBackup():
    try:
        starttime=time.time()                       #备份前的时间戳
        fsrc=open('D:\\movie.mp4','rb')
        fdest=open('F:\\a.mp4','wb+')
        print '文件备份中...'
        content=fsrc.read()
```

```
        fdest.write(content)
        print '文件备份成功'
        finishtime=time.time()                    #成功备份后的时间戳
        totaltime=finishtime - starttime          #两时间戳相减即为备份所用时间
        print totaltime
    except IOError,e:
        print e
    finally:
        fsrc.close()
        fdest.close()

testFileBackup()
```

程序先导入 time 模块，以使用其 time 函数获取当前时间戳，然后调用 time 函数记录备份前的时间戳，接着打开 D 盘下的 a.mp4 文件，并在 E 盘下创建 a.mp4 文件，然后读取 D 盘下 a.mp4 文件内容，将其写到 E 盘下刚创建的 a.mp4 文件，写完数据后再次调用 time 函数记录成功备份后的时间戳，两时间戳相减即为备份所用时间，并将其输出，程序运行结果如下：

```
文件备份中...
文件备份成功
备份所需时间(单位:s): 6.88899993896
```

可以看到，备份大小为 456MB 的文件所需的时间约为 6.9s。当然，这和系统的硬件以及备份的位置有关。

除了这样方式之外，还有其他更简便、高效的文件备份的方式吗？答案是肯定的。在 Python 中提供了 shutil 模块，这个模块是一个高级的文件操作工具，其提供的函数可以便捷、高效地实现文件的备份等操作。表 10-2 列举了该模块下的一些常用函数。

表 10-2　shutil 模块的常用函数

函　数	说　明
copyfile(src,dest)	从源 src 复制到 dest 中去，前提是目标地址具备可读写权限。如果当前 dest 已存在就会被覆盖
copymode(src,dest)	只复制其权限，其他内容不会被复制
copystat(src,dest)	复制权限、最后访问时间、最后修改时间
copy(src,dest)	复制一个文件到另一个文件或另一个目录
copy2(src,dest)	复制内容的同时也复制文件最后访问时间与修改时间
copytree(olddir,newdir,true/false)	把 olddir 复制一份 newdir，如果第三个参数是 true，则复制目录时将保持文件夹下的符号连接；如果第三个参数是 false，则将在复制的目录下生成物理副本来替代符号连接
rmtree(dirname)	删除 dirname 文件夹(可以是非空文件夹)

我们以 copyfile 函数为例来说明文件的备份，该函数有两个参数，其中 src 表示需要复制文件的地址，参数 dest 表示文件复制的目的地址。使用该函数时需要手动导入 shutil 模块。为与前面介绍的文件备份方法相比较，下面这个例子也是把 D 盘下文件大小为 456MB 的 a.mp4 文件备份到 E 盘下。

```
#coding:utf-8
#例 10-10  使用 copyfile 函数实现文件备份
import shutil
import time
def testFileBackup():
    try:
        starttime=time.time()
        shutil.copyfile('D:\\movie.mp4','F:\\a.mp4')
        print '文件备份成功'
        finishtime=time.time()
        totaltime=finishtime - starttime
        print '备份所需时间(单位:s):',totaltime
    except Exception,e:
        print e

testFileBackup()
```

程序运行结果如下：

```
文件备份成功
备份所需时间(单位:s): 0.68700003624
```

可以看到，通过 copyfile 函数备份相同文件，且备份地址也相同，其所需的时间仅约为 0.69s，与通过 read 和 write 函数实现的文件备份所需时间整整相差一个数量级。

10.5.2 文件的删除

文件的删除也是很常用的文件操作，因为在系统使用的过程中，不可避免会产生一些对我们来说用处不大的文件，有时还会因为不正当的操作而产生对系统有害的文件。这就需要文件删除操作，这样既可以消除其对系统的危害，而且也会减少硬盘空间的占用，方便操作人员对文件进行管理。

在介绍文件删除之前，我们先看看删除文件需要用到的 os 模块。Python 标准库中的 os 模块包含了普通的操作系统功能。该模块提供了统一的操作系统接口函数，这些接口函数通常是指定了操作平台的，os 模块能在不同操作系统平台下在特定函数间进行自动切换，从而能实现跨平台操作。

表 10-3 列出了 os 模块(包括子模块 path)中比较常用的函数。

删除文件需要用到其中的 remove 函数，该函数含有一个参数，表示所要删除文件的地址，使用该函数时，需要先把 os 模块导入到当前程序中。

下面通过一个例子来说明文件的删除。

表 10-3　os 模块的常用函数

函　　数	说　　明
os. getcwd()	得到当前工作目录,即当前 Python 脚本的目录路径
os. listdir(dirname)	列出 dirname 下的目录和文件
os. remove(name)	删除 name 文件
os. chdir(dirname)	把工作目录改为 dirname
os. mkdir(dirname)	创建一级目录的文件夹
os. rmdir(dirname)	删除非空文件夹
os. makedirs(dirname)	创建多级目录的文件夹
os. path. isdir(name)	判断 name 是不是一个目录,不是则返回 false
os. path. exists(name)	判断 name 文件或目录是否存在
os. path. isfile(name)	判断 name 是不是一个文件,不是则返回 false
os. path. getsize(name)	获得文件大小,如果 name 是目录则返回 0L

```
#coding:utf-8
#例 10-11　文件的删除
import os
def testFileRemove():
    try:
        filename='F:\\a.mp4'
        isExist=os.path.exists(filename)
        if isExist:
            os.remove(filename)
            print '文件删除成功'
        else:
            print '所要删除的文件不存在'
    except Exception,e:
        print e

testFileRemove()
```

该程序首先调用 os 的子模块 path 的 exists 函数判断所删除的文件是否存在,如果存在则将其删除,否则提示所要删除的文件不存在。

10.6　文件夹的创建和删除

相信有用过电脑的人都会知道怎么创建和删除文件夹,如果通过程序来创建和删除文件夹又会是怎么样的呢?下面将分别介绍在 Python 中是如何通过程序来实现文件夹的创建和删除的。

10.6.1 文件夹的创建

创建文件夹可以通过 mkdir 函数或者 makedirs 函数来实现，这两个函数的区别在于调用一次 mkdir 函数只能创建一个一级目录的文件夹，即文件夹里不能含有子文件夹，而 makedirs 函数则可以一次创建多级目录的文件夹。这两个函数都是 os 模块的函数。

下面通过一个例子来说明文件夹的创建。

```
#coding:utf-8
#例 10-12  文件夹的创建
import os
def testCreateDir():
    try:
        dirname='C:\\test'
        multipledirname='C:\\first\\second\\third'
        isExist=os.path.exists(dirname)
        if isExist:
            print dirname+'文件夹已存在'
        else:
            os.mkdir(dirname)
            print '调用 mkdir 函数成功创建一级目录的文件夹'

        isExist=os.path.exists(multipledirname)
        if isExist:
            print multipledirname+'文件夹已存在'
        else:
            os.makedirs(multipledirname)
            print '调用 makedirs 函数成功创建多级目录的文件夹'

    except Exception,e:
        print e

testCreateDir()
```

该程序先调用 exists 函数判断 C 盘下是否有 test 文件夹，如果有则提示该文件夹已存在，否则调用 mkdir 函数创建 test 文件夹。对于创建多级目录的文件夹，同样先判断其是否存在，不存在才调用 makedirs 函数将其创建。注意，如果文件夹已存在，但还是调用 mkdir 或者 makedirs 函数试图再次创建会抛 WindowsError 异常。

10.6.2 文件夹的删除

删除文件夹可以通过 rmdir 函数或者 rmtree 函数来实现，这两个函数的区别在于 rmdir 函数只能删除空的文件夹，而 rmtree 函数则可以删除非空的文件夹。此外，rmdir 函数是 os 模块的函数，而 rmtree 函数是 shutil 模块的函数。

下面通过一个例子来说明文件夹的删除。

```
#coding:utf-8
#例 10-13　文件夹的删除
import os
import shutil
def testRemoveDir():
    try:
        dirname='C:\\test'
        multipledirname='C:\\first\\second\\third'
        isExist=os.path.exists(dirname)
        if isExist:
            os.rmdir(dirname)
            print '调用 rmdir 函数成功删除空文件夹'
        else:
            print dirname+'文件夹不存在'

        isExist=os.path.exists(multipledirname)
        if isExist:
            shutil.rmtree(multipledirname)
            print '调用 rmtree 函数成功删除非空文件夹'
        else:
            print multipledirname+'文件夹不存在'

    except Exception,e:
        print e

testRemoveDir()
```

该程序先调用 exists 函数判断 C 盘下是否有 test 文件夹，如果没有则提示该文件夹不存在，否则调用 rmdir 函数删除 test 文件夹（如果 test 文件夹是非空的，会抛 WindowsError 异常）。对于删除非空文件夹，同样先判断其是否存在，存在才调用 rmtree 函数将其删除。注意，如果文件夹不存在，但还是调用 rmdir 或者 rmtree 函数试图再次删除也会抛 Windows Error 异常。

10.7　本章小结

本章主要讲解了以下几个知识点。

（1）文件。文件是指存储在外部介质上的数据的集合，文件可以看作是一个字符（字节）的序列，即由一个个字符（字节）的数据顺序组成。根据数据的组织形式，可分为 ASCII 文件和二进制文件。ASCII 文件又称为文本文件，它的每一个字节放一个 ASCII 代码，代表一个字符。二进制文件是把内存中的数据按其在内存中的存储形式原样输出到磁盘上存放。

（2）文件的打开与关闭。打开文件是使用 open 函数实现的，使用该函数时通常要指

定打开的方式,如“w”表达只能向打开的文本文件写入内容,“rb”表示只能读取文件的内容等。注意,文件也可以以二进制的方式打开。关闭文件可以使用 close 函数实现,应该养成在程序终止之前关闭所有文件的习惯,否则可能会丢失数据。

(3) 文件的读写。文件的读取有三个函数,它们分别是 read()、readline()和 readlines()函数。其中,read()函数可一次性读取位置指针后面的所有数据,readline()函数是按行读取数据,而 readlines()函数则以多行的方式一次性读取位置指针后面的所有数据。将内容写入到文件中有两个函数,它们分别是 write()和 writelines()函数。其中,write()函数是把一个字符串写入到文件中,而 writelines()函数则是把列表中的字符串内容写入到文件中。

(4) 文件的定位。文件中有一个位置指针,指向当前读写的位置。如果顺序读写一个文件,每次读写一个字符,则读写完一个字符后,该位置指针自动移动指向下一个字符。可以使用 seek 函数改变文件的位置指针,从而实现随机读写文件的功能。此外,tell 函数可以得到位置指针的当前位置。

(5) 文件的备份和删除。文件的备份可以通过文件的读写操作实现,但这种方式比较麻烦,更重要的一点是效率很低,而通过 shutil 模块的函数进行文件备份,操作简便,而且效率也非常高。文件的删除可以通过 os 模块的 remove 函数实现,但通常先调用 os 的子模块 path 的 exists 函数判断所删除的文件是否存在。

(6) 文件夹的创建和删除。文件夹的创建可以通过 os 模块的 mkdir 函数或者 makedirs 函数来实现。其中,调用一次 mkdir 函数只能创建一个一级目录的文件夹,即文件夹里不能含有子文件夹,而 makedirs 函数则可以一次创建多级目录的文件夹。文件夹的删除可以通过 os 模块的 rmdir 函数或者 shutil 模块的 rmtree 函数来实现,其中 rmdir 函数只能删除空的文件夹,而 rmtree 函数则可以删除非空的文件夹。

10.8 习　　题

一、解答题

1. 什么是文件?文件分为哪几类?它们的特点是什么?

2. 文件的打开和关闭分别用到什么函数?打开方式有哪些?它们的含义分别是什么?

3. 读取文件有哪些函数?它们的区别是什么?写入文件又有哪些函数?它们的区别又是什么?

4. 备份文件有哪些方式?它们的优缺点各是什么?如何删除一个文件?

5. 创建文件夹有哪些函数?它们的区别是什么?删除文件夹又有哪些函数?它们的区别又是什么?

二、看程序写结果

1. 假设 D 盘的 a.txt 文件有三行数据,第一行:abc(有换行符) 第二行:def(有换行

符）第三行：ghi

```
def testRead():
    try:
        f=open('D:\\a.txt','r+')
        content=f.read(1)
        print content
        content=f.readline()
        print content
        content=f.readlines()
        for con in content:
            print con

    except IOError,e:
        print e
    finally:
        f.close()

testRead()
```

2.

```
def testReadAndWrite():
    try:
        f=open('D:\\a.txt','w+')
        f.write('123')
        f.writelines(['abc\n','def'])
        print f.tell()
        f.seek(0)
        content=f.readlines()
        for con in content:
            print con

    except IOError,e:
        print e
    finally:
        f.close()

testReadAndWrite()
```

3.

```
def testReadAndWrite():
    try:
        f=open('D:\\a.txt','w+')
        f.writelines(['abc\n','def\n','ghi'])
```

```
            f.seek(-3,2)
            print f.tell()
            content=f.read()
            print content
            f.seek(-6,1)
            f.write('123')
            f.seek(0,0)
            content=f.read()
            print content
        except IOError,e:
            print e
        finally:
            f.close()

testReadAndWrite()
```

三、上机练习

1. 输入一个字符串,将其中的小写字母转为大写字母,然后写入 test.txt 文件中(提示:可以使用 upper 函数)。

2. 有两个文件 a.txt 和 b.txt,它们各存放一行按字母表顺序排好的小写字母,现在将它们的内容合并起来,并保证合并的内容按字母表顺序排好,然后将它们写入 c.txt 文件中。

3. 提示输入一个文件名,如果存在,再提示输入一个数字 n,然后显示该文件的前 n 行的内容,如果输入的文件名不存在,则提示文件不存在。

4. 写一个比较两个文件的程序,如果两文件不同,给出第一个不同处的行号和列号,并分别输出不同的字符是什么,如果都相同,则输出两文件相同。

5. 写一个日志记录和查看的程序,启动程序时提示输入数字选择记录日志、查看日志还是退出程序。如果记录日志,在输完内容后,提示输入文件名,创建以该文件名命名的文件,并把内容保存到该文件中;如果查看日志,先列出当前文件夹下所有的文件名(假设当前文件夹只存有日志文件),然后提示输入要查看的日志文件,最后把内容输出。如图 10-2 所示。(提示:列出当前文件夹下所有的文件名需要用到 os.path. walk(path,func,args)函数,path 参数代表需要访问的文件夹的绝对地址,func 参数代表回调函数,args 参数代表需要传给 func 函数的参数。func(args,path,re sult)函数通常用于对文件名列表进行处理,args 和 path 参数分别对应 walk 函数中的 args 和 path 参数,result 参数是 path 目录下所有文件名组成的列表)。

6. 将 C 盘 a 文件的 a.txt 文件备份到 b 文件夹下,然后删除 a 文件的 a.txt 文件。

7. 创建一个一级目录的 single 文件夹和一个多级目录的 multiple/second/t hird 文件夹,然后又删除 single 文件夹和 multiple 文件夹(包括子文件夹)。

```
Python 2.7.10 Shell
File Edit Shell Debug Options Window Help
请输入数字选择功能（1：记录日志，2：查看日志，3：退出程序）1
请输入日志内容
2015年12月5日，我学习了Python的命名空间。命名空间是从名称到对象的映射。Python中有三类命名空间
：内建命名空间、全局命名空间和局部命名空间。在不同的命名空间中的名称是没有关联的。
请输入保存的文件名
2015-12-5
保存成功

请输入数字选择功能（1：记录日志，2：查看日志，3：退出程序）1
请输入日志内容
2015年12月15日，我学习了Python的异常。异常是程序运行过程中由于出现语法错误或逻辑错误而发生的事
件，该事件可以中断程序指令的正常执行流程。
请输入保存的文件名
2015-12-15
保存成功

请输入数字选择功能（1：记录日志，2：查看日志，3：退出程序）1
请输入日志内容
2016年1月5日，我学习了Python的文件。文件是指存储在外部介质上的数据的集合，文件可以看作是一个字
符（字节）的序列，即由一个个字符（字节）的数据顺序组成。根据数据的组织形式，可分为ASCII文件和二进
制文件。ASCII文件又称为文本文件，它的每一个字节放一个ASCII代码，代表一个字符。二进制文件是把内存
中的数据按其在内存中的存储形式原样输出到磁盘上存放。
请输入保存的文件名
2016-1-5
保存成功

请输入数字选择功能（1：记录日志，2：查看日志，3：退出程序）2
当前目录的文件列表如下：
2015-12-15
2015-12-5
2016-1-5
请输入要查看的文件名
2015-12-15
2015-12-15文件的内容如下：
2015年12月15日，我学习了Python的异常。异常是程序运行过程中由于出现语法错误或逻辑错误而发生的事
件，该事件可以中断程序指令的正常执行流程。

请输入数字选择功能（1：记录日志，2：查看日志，3：退出程序）3
退出程序
>>>
```

图 10-2　上机题第 5 题示意图

第 11 章

项目开发实例

本章学习目标

- 理解 Django 框架
- 理解 MVC 模式和 Django 的 MTV 模式
- 掌握 Django 的安装
- 能够部署运行本章的案例

经过前面章节的介绍，相信读者已经对 Python 有了比较全面的认识。在本章我们将结合一个项目开发的实例介绍 Python 应用于 Web 的应用程序。本章的项目是在一个通用的 Python 框架——Django 上开发的实例，本章内容需要读者具备基本的 HTML，CSS，JavaScript 的知识和 SQL 原理的知识。

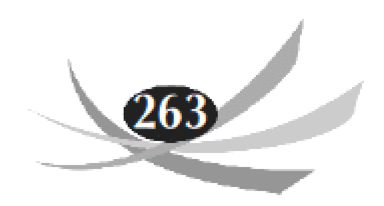
项目中引入并集成了一些优秀的开源框架，如前端框架 Bootstrap，若读者对这些框架完全不了解并不影响对本章项目的理解。

11.1 Django 框架简介

Django 之于 Python 如同 Zend Framework 之于 PHP，.NET Framework 之于 C＃。Django 是由 Python 开发的应用于 Web 开发的免费开源的高级动态语言框架，在全球拥有一个活跃度很高的社区。它最初是被开发用于管理劳伦斯出版集团旗下的一些以新闻内容为主的网站的，并于 2005 年 7 月在 BSD(Berkeley Software Distribution，伯克利软件套装)开放源代码协议许可下授权给开发者自由使用。这套框架是以比利时的吉普赛爵士吉他手 Django Reinhardt 来命名的。使用 Django，我们在很短的时间内就可以创建出高品质、易维护、数据库驱动的应用程序。

当前基于互联网的开源特性，我们常提到的 LAMP(Linux Apache Mysql/MariaDB Python/PHP)中的 P 一般是指 Python 或者 PHP，而针对 Python 的 Django 和 PHP 的 Zend Framework 两种各自阵营内最为活跃的框架技术，谁优谁劣不在本书的讨论范围，作者曾在 Zend Framework 上做过大量的项目开发，目前也开始转向使用 Django 开发新的项目。

Django 拥有完善的模板(Template)机制、对象关系映射(Object- Relation Mapper)机制以及拥有动态创建后台管理界面的功能。使用 Django 框架来开发 Web 应用，可以快速设计和开发具有 MVC(Model View Controller)层次的 Web 应用。Django 框架是从实际项目中诞生出来的，该框架提供的功能特别适合于动态网站的建设，特别是管理接口。



在 Django 框架中，包含了 Web 开发网络应用所需的组件。这些组件包括数据库对象关系映射、动态内容管理的模板系统和丰富的管理界面。在 Django 框架中，可以使用管理脚本文件 manage.py 来构建简单的开发服务器。

Django 框架作为一个快速的 Web 应用开发框架，具有下面的一些特点。

- 丰富的组件：在 Django 框架中，有丰富的用于开发 Web 应用的组件，这些组件都是用 Python 开发的，并为开源界所修改和使用。Django 框架中的组件的设计目的是实现重用性，并具有易用性。
- 对象关系映射和多数据库支持：Django 框架的数据库组件——对象关系映射提供了数据模块和数据引擎之间的接口。支持的数据库包括 PostgreSQL、MySQL 和 SQLite 等。这种设计使得在切换数据库的时候只需要修改配置文件即可。这为应用开发者在设计数据库时提供很好的灵活性。
- 简洁的 URL 设计：Django 框架中的 URL 系统设计非常强大而灵活。可以在 Web 应用中为 URL 设计匹配模式，并用 Python 函数处理。这种设计使得 Web 应用的开发者可以创建友好的 URL，且更适合于搜索引擎的搜索。
- 自动化的管理界面：在 Django 框架中已经提供了一个易用的管理界面，这个界面可以方便地管理用户数据，具有高度的灵活性和可配置性。
- 强大的开发环境：在 Django 中提供了强大的 Web 开发环境，其中有一个可用于开发和测试的轻量级 Web 服务器。当启用调试模式后，Django 会显示大量的调试信息，使得消除 bug 非常容易。

11.2 MVC 模式

MVC 是指现代程序设计中的一种分层设计模式，将传统程序按其功能边界分为表现层、逻辑层和控制层三部分，这种方式的划分能使程序设计变得更加容易，缩短了程序开发周期，同时开发出来的程序也易于维护。

11.2.1 MVC 的概念

MVC 是 Model（模型）、View（视图）和 Controller（控制器）的缩写，它是一种在软件工程中广泛使用的设计模式，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

MVC 模式直观上就是将写在一个文件中的代码按功能分别写在了三个文件中，不同的功能由不同的文件去具体实现。可以设想以下场景：

- 用户希望调整界面布局，如把各栏目的位置和颜色搭配做适当的调整，以尽量消除出现的审美疲劳。
- 用户希望按访问量由高到低的顺序显示新闻列表而不是根据发布时间来排序。
- 用户希望更换数据库以达到更好的稳定性。

在传统的开发模式中，有可能需要三个角色来参与上述业务变动而导致的修改工作，

但由于是写在一个文件中，因此只能是依次修改，效率显得很低，并且这种情况容易导致后面修改的人不小心改到别人的代码。而在 MVC 模式中，虽然还是要三个角色参与，但由于文件是分开的，因此可以同时修改，大大提高了开发效率，并且还不用担心其他人是否会不小心误改了自己的代码。因此 MVC 模式带来的最大好处就是明晰了整个程序中各个功能代码片段的逻辑分界，使 Web 应用程序开发具有更好的可维护性与复用性，同时使得团队开发具有更高的规范性和可控性。

MVC 模式的核心是将功能完整的程序分成了业务逻辑上可独立的三个部分：视图、模型、控制器。

视图一般是指程序运行的用户界面(Users Interface，UI)，是用户对系统功能最直观的体验。在相同的业务逻辑下，可能存在不同的视图。例如，决策者喜欢用图表来呈现数据，而技术人员则偏向于用统计报表来展现数据。视图的外观样式给人的感受也会因人而异，所以视图是三者中最面向用户、最易变的部分。视图上动态显示的数据来自于模型。

模型一般是指实体或业务逻辑本身，在 PHP 的 MVC 中，模型是指业务数据的存储及与之相关的业务处理。如客户数据，能被多个视图所共享，并且带有业务处理逻辑，可根据情况为不同的视图提供适当的数据。模型除了体现为表对象模型外，还能支持数据持久化操作，也可以包含相应的业务处理规则，非常灵活。模型负责为视图提供结果数据，同时也负责交互式系统中用户输入数据的持久化操作。

控制器一般是指连接视图与模型的纽带。视图所需要的数据由控制器通知相应的模型来提供，控制器也负责一定范围内的逻辑处理，如对有效性的检测，或是如果模型本身不提供的业务处理，则可由控制器来处理。

简单地说，模型负责把数据从数据库中取出或存入数据库，控制器则根据浏览器的 URL 地址访问“模型”获取数据，并调用“视图”显示这些数据，最后视图将数据格式化后呈现给用户。控制器将模型和视图隔离，作为它们连接的中间桥梁。如图 11-1 所示。

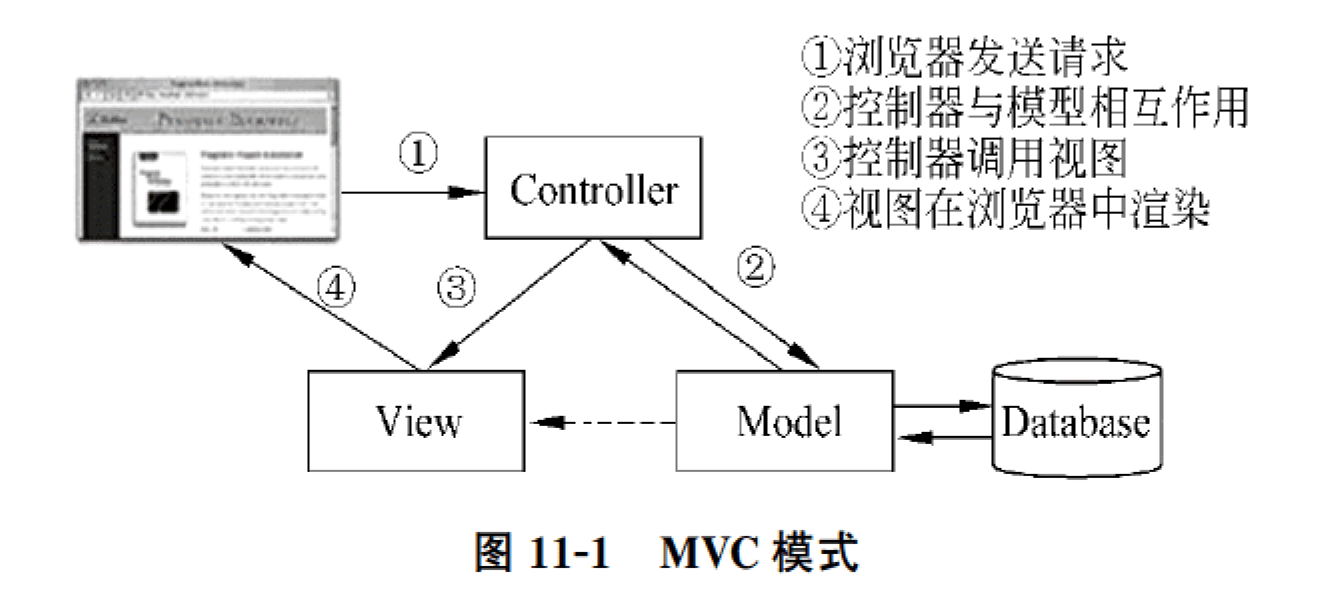
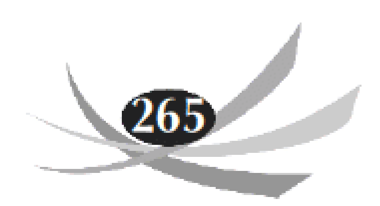


图 11-1 MVC 模式

11.2.2 Django 的 MTV 模式

Django 作为一个流行的基于 Python 的 Web 开发框架，也支持 MVC 模式。在 Django 框架中，当发生 URL 请求时，将会调用指定的 Python 方法。通过业务逻辑处理后，将会通过模板来呈现页面。Django 更关注的是模型(Model)、模板(Template)和视图(View)，称为 MTV 模式：



M 代表模型,即数据存取层。该层处理与数据相关的所有事务:如何存取数据、如何验证数据的有效性以及如何描述数据之间的关系等。在这一层中,Django 使用 django.db.model.Model 实现了网站设计中需要使用的数据模型,在数据模型中定义了保存在数据库中的各种对象及其属性。通过继承自 Model 类生成的对象,可以通过添加 Field 来为特定的数据增加方法。在 Django 数据模型中,提供了丰富的访问数据对象接口。数据模型中的数据将会同步到后台的数据库中,Django 提供了一个良好的对象关系映射,使得开发者可以从视图和模板中访问数据库中的数据。

T 代表模板,即表现层。该层处理与表现相关的逻辑:如何在页面或其他类型文档中进行显示等。Django 提供了强大的模板解析功能,通过页面函数来输出页面响应。模板系统使得 Web 应用的开发者可以把注意力集中在需要展现的数据上,而页面设计者只需关注输出页面的构成。

V 代表视图,即业务逻辑层。该层包含存取模型及调取恰当模板的相关逻辑。可以把它看作模型与模板之间的桥梁。在这一层,Django 框架实现了良好的 URL 设计和处理。当收到 URL 请求时,Django 将会使用一组预订的 URL 模式来匹配合适的处理器。实际上,URL 的设计也是网站视图层设计,决定了 Web 应用如何读取 URL 请求以及如何显示网页。对每个特定的 URL,Django 都会有一个特定的视图函数来进行处理。可以看到,Django 框架的视图处理分成多个步骤。其框架在收到 URL 请求后,通过页面函数来处理,最后将页面响应返回给浏览器显示。

11.3　Django 安装

由于 Python 语言的跨平台性,因此 Django 可以很方便地安装在 Windows 和 Linux 等系统平台上。这里仍以 Windows 系统为例来介绍 Django 的安装过程。安装步骤如下:

(1) 到 Django 的官网 https://www.djangoproject.com/下载 Django 压缩包。目前 Django 官方最新版本是 1.9.1。

(2) 解压 Django-1.9.1.tar.gz 压缩包,这里解压到 C 盘的 Django-1.9.1 目录。

(3) 打开 DOS 命令窗口,执行 cd 命令转到 Django-1.9.1 目录下。这里执行的命令是 cd \Django-1.9.1。

(4) 执行 setup.py install 命令,启动 Django 安装程序。安装成功后,可以通过执行命令验证 Django 是否安装成功。

```
>>>import django
>>>django.get_version()
'1.9.1'
```

输出 Django 的版本号 1.9.1,则说明 Django 已安装成功。如图 11-2 所示。

除了以上这种方式外,还可以用包管理工具 pig 来安装,并且这种安装方式更为简便。

Django 作为一个第三方框架,我们可以把其当作一个包,因此可以采用 pip 包工具安

```
C:\WINDOWS\system32\cmd.exe - python
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\wangs>python
Python 2.7.10 (default, May 23 2015, 09:44:00) [MSC v.1500 64 bit (AMD64)] on wi
n32
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> django.get_version()
'1.9.1'
>>>
```

图 11-2　测试 Django 安装结果

装 Django，在安装 Django 前要确保操作的电脑已安装 pip 包管理工具，关于如何安装 pip 包管理工具请参考 8.3.2 节。

打开 DOS 命令窗口，执行以下命令：

```
pip install Django==1.9.1
```

当安装成功后会有如下的提示信息：

```
Installing collected packages: Django
Successfully installed Django-1.9.1
```

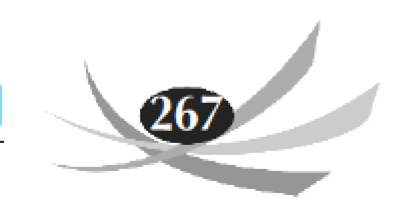
pip 安装 Django 的结果如图 11-3 所示。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\wangs>pip install Django==1.9.1
Collecting Django==1.9.1
  Downloading Django-1.9.1-py2.py3-none-any.whl (6.6MB)
    100% |████████████████████████████████| 6.6MB 64kB/s
Installing collected packages: Django
Successfully installed Django-1.9.1

C:\Users\wangs>
```

图 11-3　pip 安装 Django 结果



11.4　创建 Django 项目

在成功安装 Django 框架后，就可以使用 Django 框架开发 Web 应用了。Django 框架提供了一种快捷的方法来创建功能丰富的 Web 应用。接下来使用 Django 创建一个简单的学生信息管理系统。

11.4.1　创建开发项目

当我们创建项目时，需要用到 django-admin. py 文件，它位于 Python 的安装根目录下的 Scripts 目录。在 DOS 命令窗口下，先转到 Script 目录，然后执行以下命令创建一个名为 xmustu 的项目：

```
django-admin.py startproject xmustu
```

使用 django-admin. py 创建 xmustu 项目如图 11-4 所示。

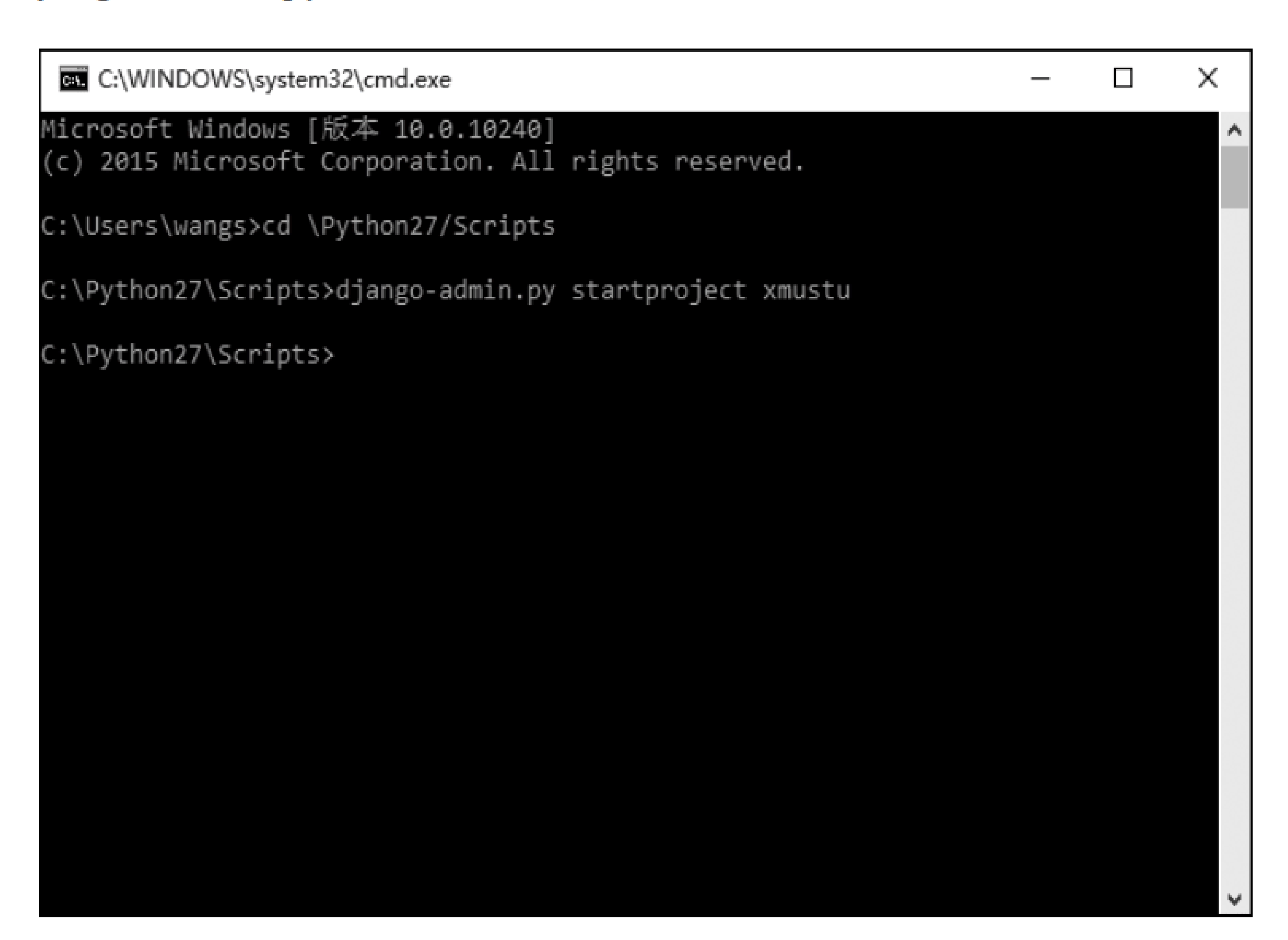


图 11-4　创建 xmustu 项目

执行以上命令后，Django 将在 Scripts 目录下自动为我们创建 xmustu 项目，并拥有如下目录结构：

```
xmustu
|--xmustu
|   |--__init__.py
|   |--settings.py
|   |--urls.py
|   |--wsgi.py
|--manage.py
```

下面对这一结构中的各文件进行说明：

- __init__. py 空文件,可以根据需要进行必要的初始化,此外还用于打包 python 工程。
- setting. py 项目的默认配置文件,包括了数据库信息、调试标识以及其他一些重要的变量。
- urls. py URL 配置文件,主要是将 URL 映射到应用程序中的相应函数。
- wsgi. py 内置 runserver 命令的 WSGI 应用配置文件。
- manage. py 是 Django 中的一个工具,用于管理 Django 站点。

xmustu 项目的目录结构如图 11-5 所示。

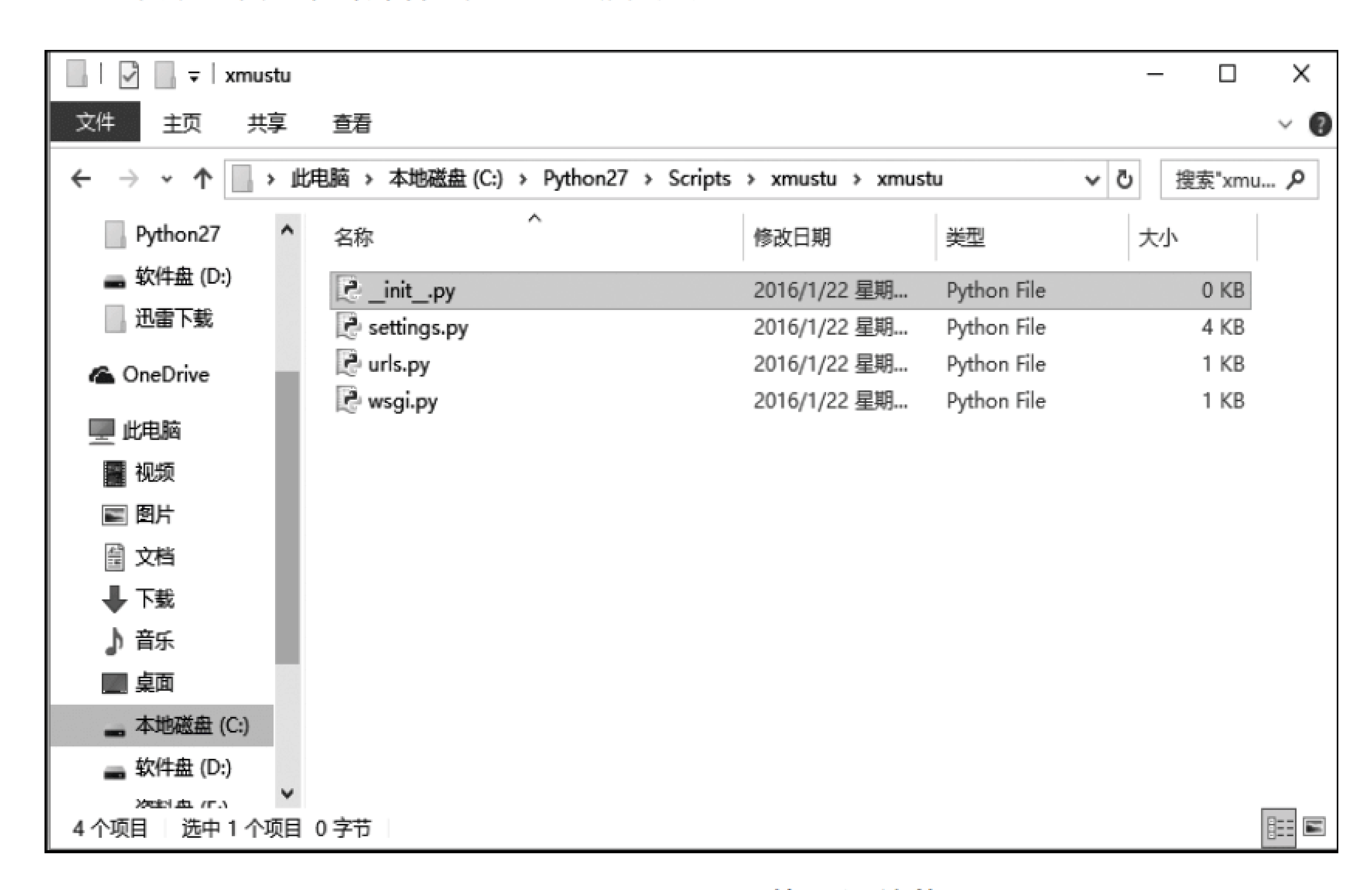


图 11-5 xmustu 项目的目录结构

11.4.2 运行开发服务器

在 DOS 命令窗口下转到 xmustu 目录(下面凡是以 manage. py 开头的命令都是在 DOS 命令窗口,xmustu 目录下),输入如下命令启动服务器:

```
manage.py runserver
```

成功启动后如图 11-6 所示。

如果启动时报错,且 errno 为 10013,这是因为 8000 端口被其他应用进程占用,这时需要关掉对应的进程,或者修改 Django 服务器监听的端口(默认是 8000),如执行以下命令监听 9000 端口。

```
manage.py runserver 9000
```

执行以上命令,只能在本机上访问该服务器,即只能接受本机的请求。在多人开发 Django 项目时,可能需要从其他主机访问该服务器,此时可以使用下面的命令来接收来自其他主机的请求:

```
manage.py runserver 0.0.0.0:8000
```

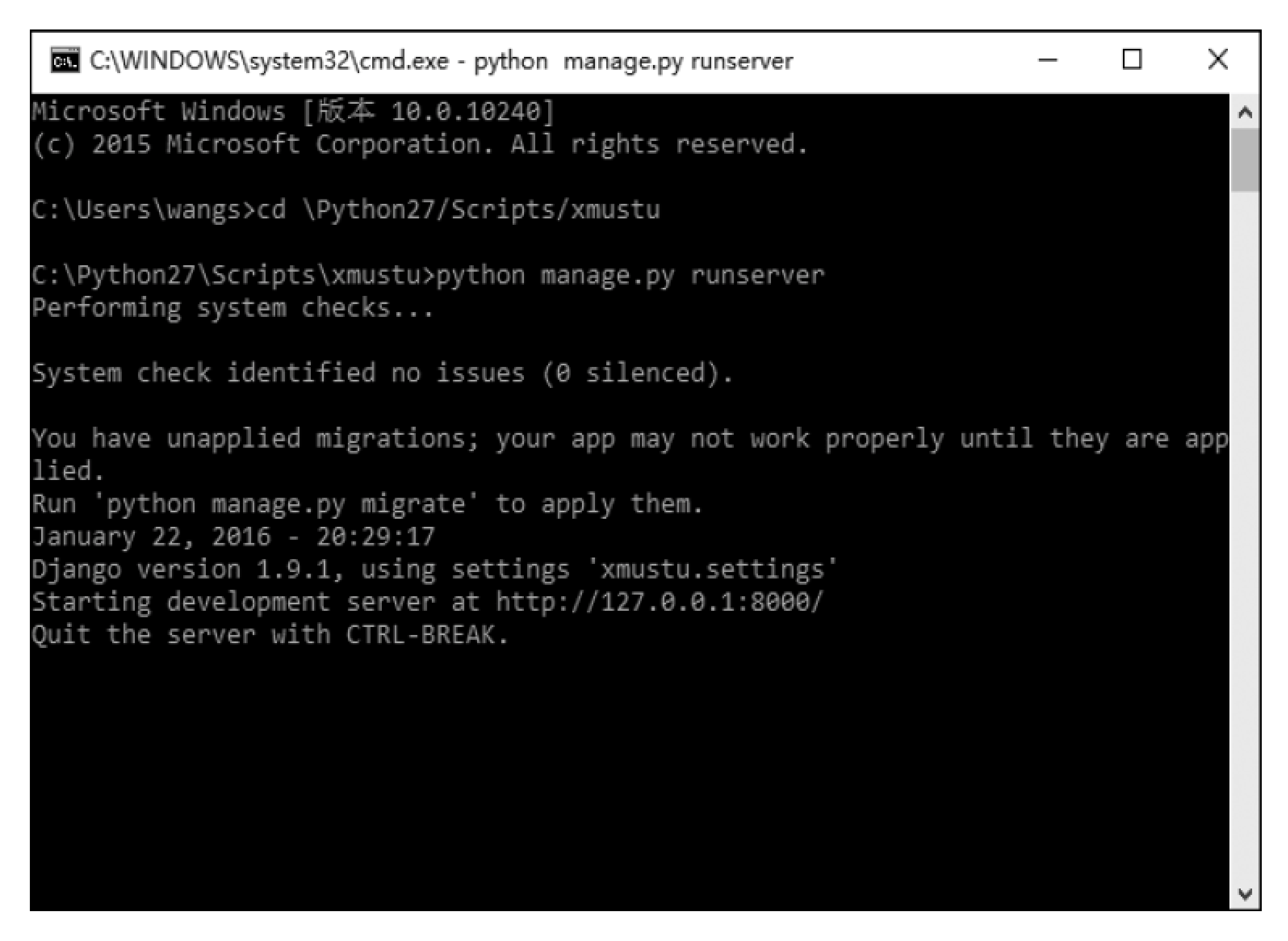


图 11-6　启动项目服务器

该语句表示服务器监听本机所有网络接口的 8000 端口，这样可以满足多人合作开发和测试 Django 项目的需求，同时也可以使用其他主机访问该服务器。

然后在浏览器中输入“http：//127.0.0.1：8000/”，将会显示 Django 项目的初始化页面，如图 11-7 所示。

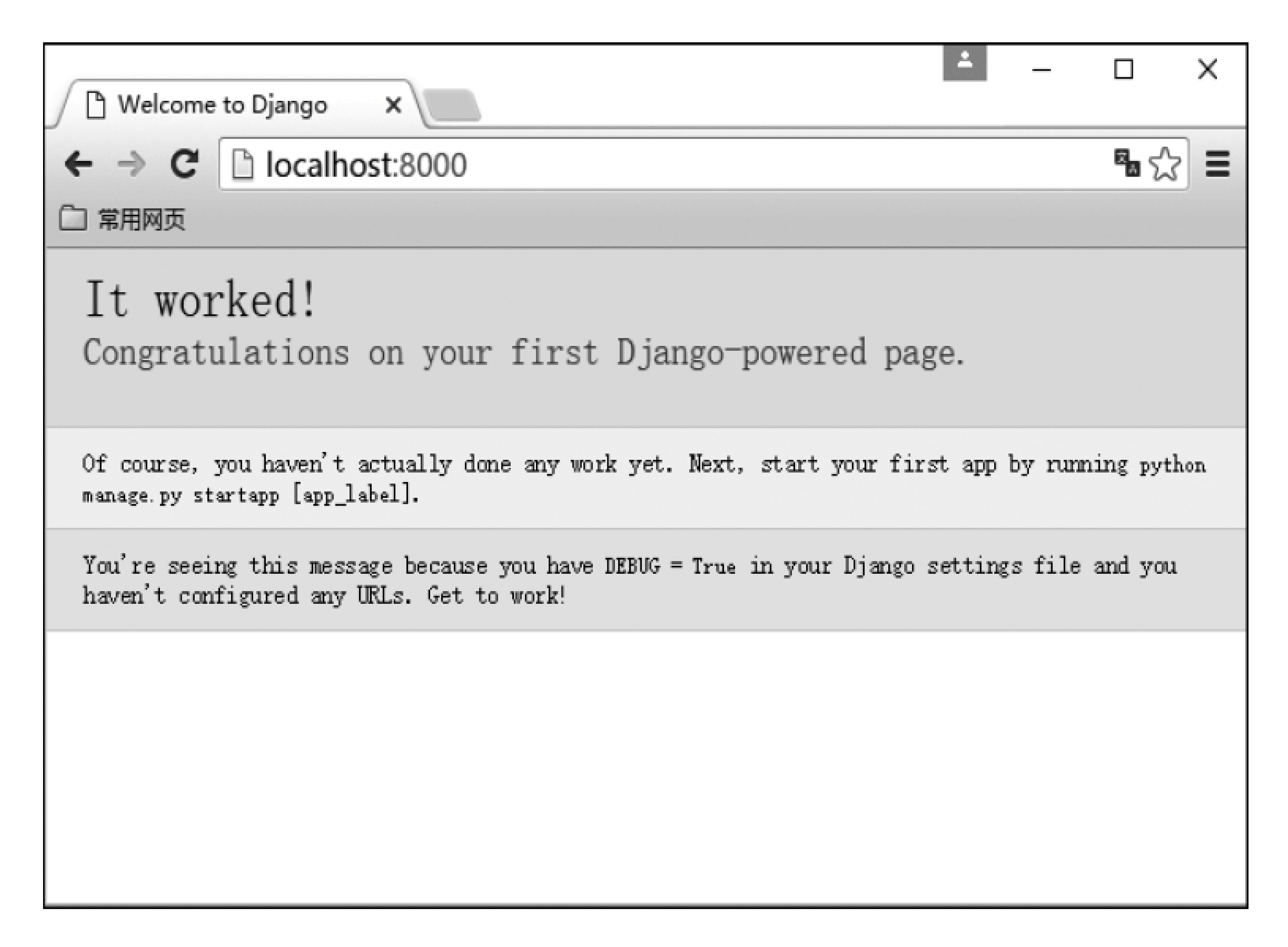


图 11-7　Django 项目的初始化页面

看到 Django 项目的初始化页面的同时，在命令窗口也会看到如下信息：

```
[22/Jan/2016 21:13:26] "GET / HTTP/1.1" 200 1767
```

此信息显示了连接的时间以及响应信息，HTTP 的状态码为 200，表示此连接已成功。

至此我们已经成功地配置并运行了一个最简单的 Django 项目。

11.5 Django 项目的高级配置

11.5.1 创建项目应用

在上一节中，我们创建了一个最基本最简单的 Django 项目，但这个项目目前并不能实现什么功能。一个 Django 项目由一个或多个应用(Application)构成，我们可以根据项目模块来划分应用。为了实现我们构建项目的功能，我们还需要建立起 MTV 框架的应用。

我们可以通过 manage. py 文件的 startapp 命令在 xmustu 项目中创建一个名为 stu 的应用，用于实现学生个人信息的相关操作业务。命令如下：

```
manage.py startapp stu
```

执行上述命令后，Django 将在 xmustu 目录下自动为我们创建 stu 应用，并拥有如下目录结构：

```
stu
|--migrations
|  |--__init__.py
|  |--admin.py
|  |--apps.py
|  |--models.py
|  |--tests.py
|  |--views.py
```

下面对这一结构中的各文件进行说明：

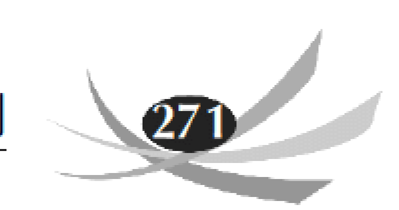
- __init__. py 空文件，用于将整个应用作为一个 Python 模块加载。
- admin. py 用于注册数据模型的文件。
- apps. py 用于配置应用的文件。
- models. py 定义数据模型相关的信息。
- tests. py 创建应用的测试文件。
- views. py 定义视图相关的信息。

stu 应用的目录结构如图 11-8 所示。

创建 stu 应用成功后需要在 xumstu 目录下的 setting. py 文件中找到 INSTALLED_APPS 元组，在其元素最后面加入'stu'，使得 Django 能够识别到已成功创建一个 stu 的应用，如图 11-9 所示。

11.5.2 配置文件

Django 的 setting. py 配置文件涉及许多的功能，由于篇幅有限，本节将介绍其中几个主要的配置。



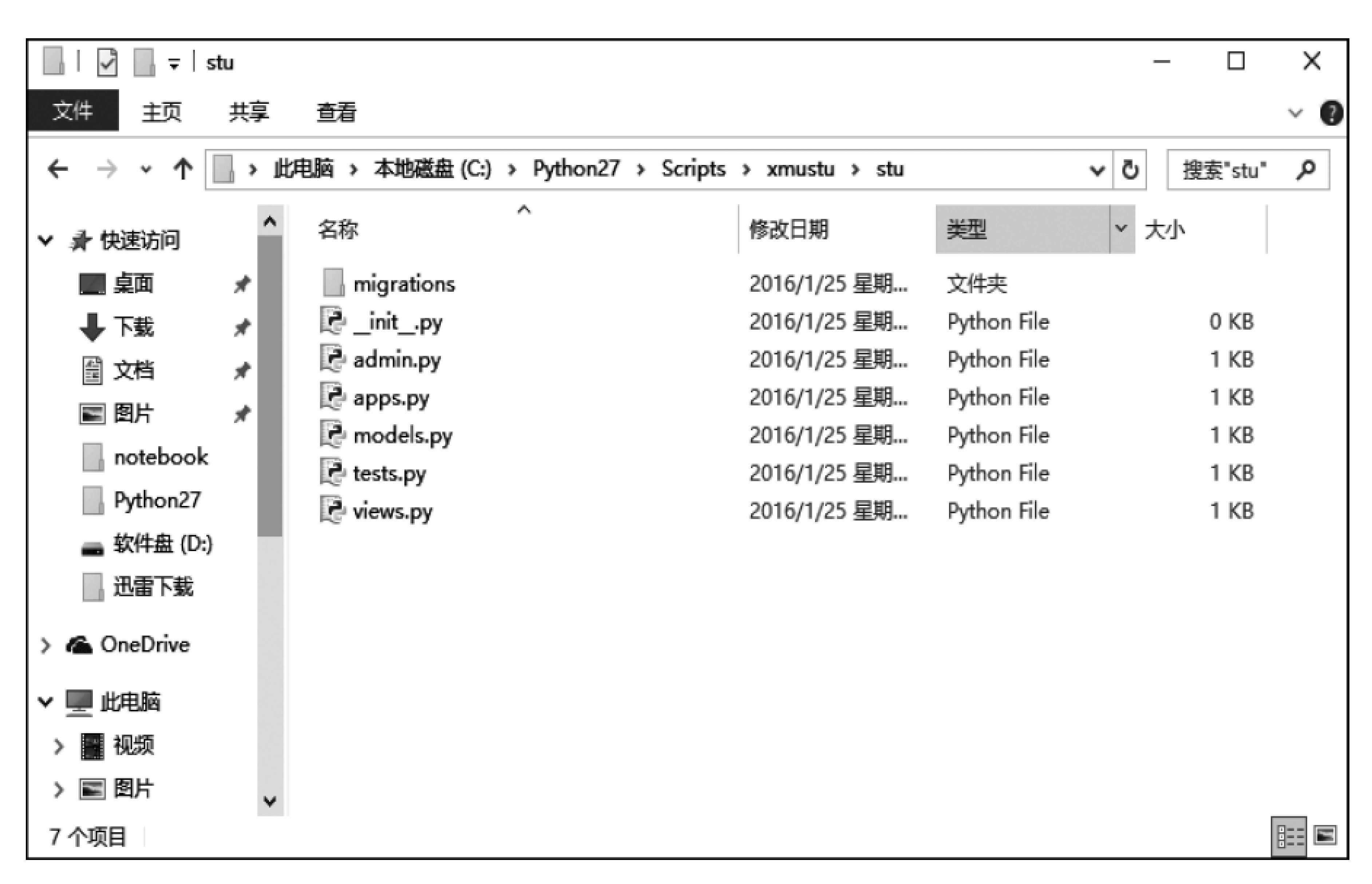


图 11-8　stu 应用的文件结构

```
C:\Python27\Scripts\xmustu\xmustu\settings.py - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 格式(M) 语言(L) 设置(T) 宏(O) 运行(R) 插件(P) 窗口(W) ?
settings.py
24
25  # SECURITY WARNING: don't run with debug turned on in production!
26  DEBUG = True
27
28  ALLOWED_HOSTS = []
29
30
31  # Application definition
32
33  INSTALLED_APPS = [
34      'django.contrib.admin',
35      'django.contrib.auth',
36      'django.contrib.contenttypes',
37      'django.contrib.sessions',
38      'django.contrib.messages',
39      'django.contrib.staticfiles',
40      'stu',
41  ]
42
43  MIDDLEWARE_CLASSES = [
44      'django.middleware.security.SecurityMiddleware',
45      'django.contrib.sessions.middleware.SessionMiddleware',
Python file  length : 3174  lines : 123  Ln : 30  Col : 1  Sel : 0 | 0  UNIX  UTF-8  INS
```

图 11-9　在 setting. py 文件的 INSTALLED_APPS 元组中加入 stu 应用

1. 开发环境与生产环境

Debug=True

所创建的项目默认是在运行开发环境中，此时，开发中所有的异常将会直接显示在网页上。例如，我们将项目搭建起来后，试图访问一个不存在的页面，将会产生 404 错误，提示网页不存在，如图 11-10 所示。

由于在开发者环境下，系统运行的所有异常将会被显示在页面上，甚至可能暴露出内部的一些数据安全信息，这是我们所不希望看到的，因此在完成开发工作并进行部署时，我们需要将开发环境转化为生产环境，并指定如何处理诸如 404，500 之类的错误。

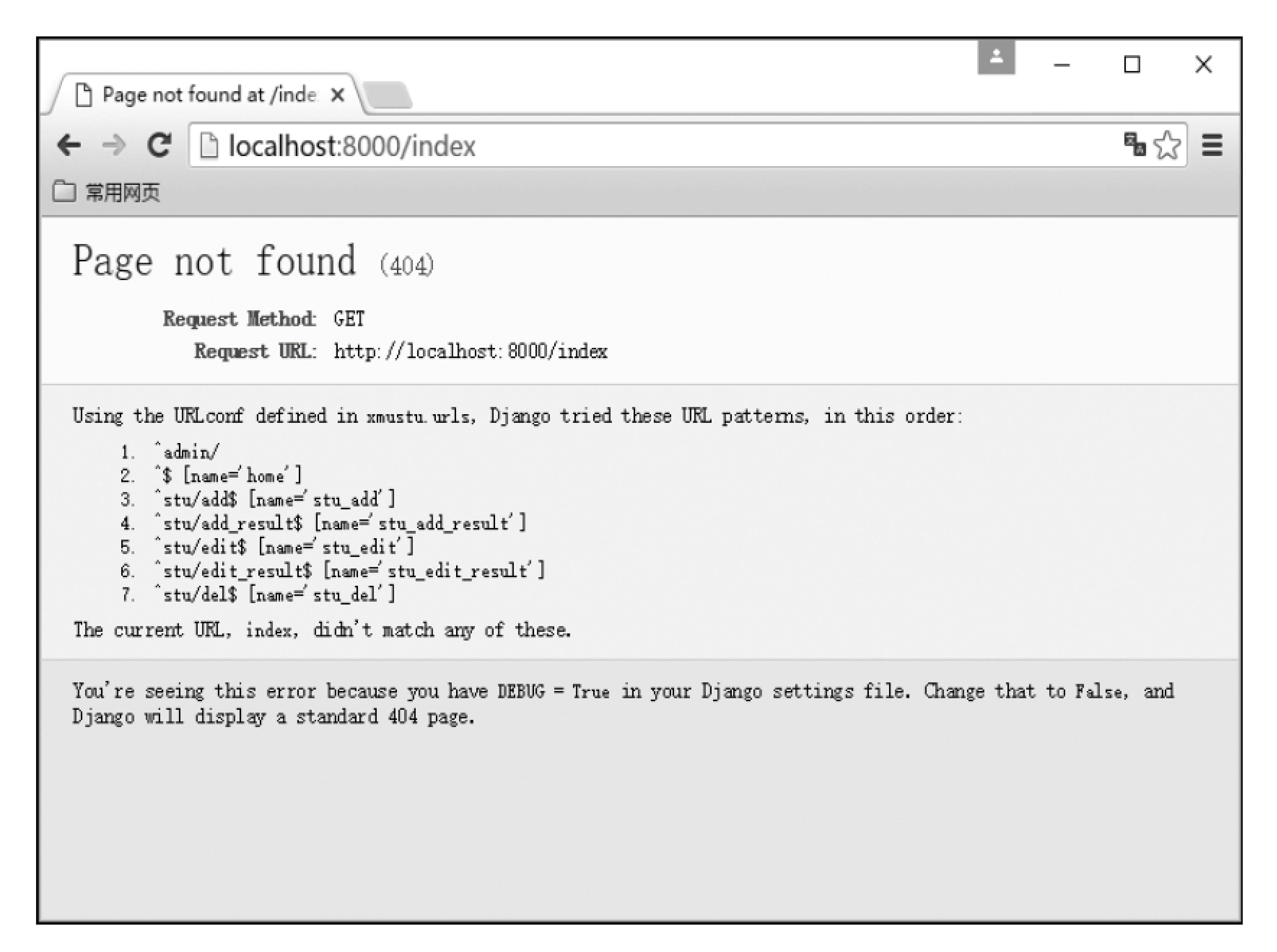


图 11-10 访问不存在的页面时的结果

生产环境中，需要设定 ALLOWED_HOSTS，以允许访问的地址，提高系统的安全性。例如，我们把 ALLOWED_HOSTS 设置为['192.168.*.*']，这样就只有局域网内的用户可以访问该项目。当发布一个公共对外的站点时，我们需要把 ALLOWED_HOSTS 设置为['*']。如下所示：

```
Debug=False
ALLOWED_HOSTS=['*']
```

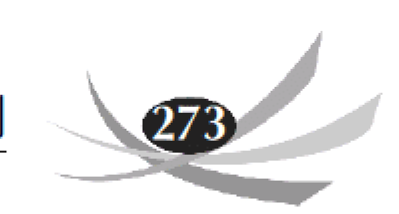
2. 配置数据库信息

在 Web 开发中，开发人员需要选择与自己所开发的项目相吻合的数据库。SQLite 数据库作为一种轻量级嵌入式的数据库引擎，有着其他数据库所不具备的优点。本章的案例将使用 SQLite 数据库引擎。

在创建 SQLite 数据库前，需要先修改 setting.py 配置文件中的 DATABASES 字典，配置相应的属性值对数据库进行设置。配置如下：

```
#Database
#https://docs.djangoproject.com/en/1.9/ref/settings/#databases

DATABASES={
    'default':{
        'ENGINE':'django.db.backends.sqlite3',
        'NAME':'db_xmu_stu',
        'USER':'',
        'PASSWORD':'',
        'HOST':'',
```




```
            'PORT': '',
        }
    }
```

其中,ENGINE用来指定使用的是SQLite3类型的数据库,NAME用来指定使用的数据库文件为db_xmu_stu,使用SQLite3数据库,其他字段的值无须设置,配置好就可以创建数据库了,创建数据库的具体内容将在11.4.4节中介绍。

3. MiddleWare中间件

Django的MiddleWare中间件的“中间”指的是服务器接受到Request到View处理,以及View处理完到发送Response给客户端这两个“中间”。熟悉Java Web开发的读者可能会发现,这个Django的中间件和Filter过滤器类似。Django的安装部署可以不需要任何的中间件,但强烈建议添加上中间件。浏览器每发送一个请求都是先通过中间件中的process_ request函数,这个函数将返回None或者HttpResponse对象,如果返回None,继续处理其他中间件,如果返回一个HttpResponse,就绕过其他中间件,返回到网页上。为了激活中间件组件,需要把它添加到settings.py配置文件中的MIDDLEWARE_CLASSES列表中,在MIDDLEWARE_CLASSES里,每个中间件组件通过一个字符串来表示。例如,下面是通过django-admin.py startproject命令创建的默认的MIDDLEWARE_CLASSES:

```
MIDDLEWARE_CLASSES=(
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
)
```

这些中间件的顺序是有意义的,在请求和视图阶段,Django使用MIDDLEWARE_CLASSES给定的顺序申请中间件,而在应答和异常阶段,Django使用相反的顺序申请中间件,即Django把MIDDLEWARE_CLASSES当作一种视图方法的“包装器”:在请求时,它自顶向下申请这个列表的中间件到视图,而在应答时它反序进行。

这里需要强调CSRF(Cross-site request forgery)中间件,Django中默认是有开启CSRF的,这样就可以防止跨站请求伪造,所以在所有的表单交互中,我们需要添加CSRF的验证,否则会产生图11-11所示的错误。

注意:防CSRF请求错误只有创建了相应的文件并配置好才能演示,这将在11.6节中说明。

11.5.3 设计数据模型

Django的Model中一般封装了与应用程序的业务逻辑相关的数据以及对数据的处

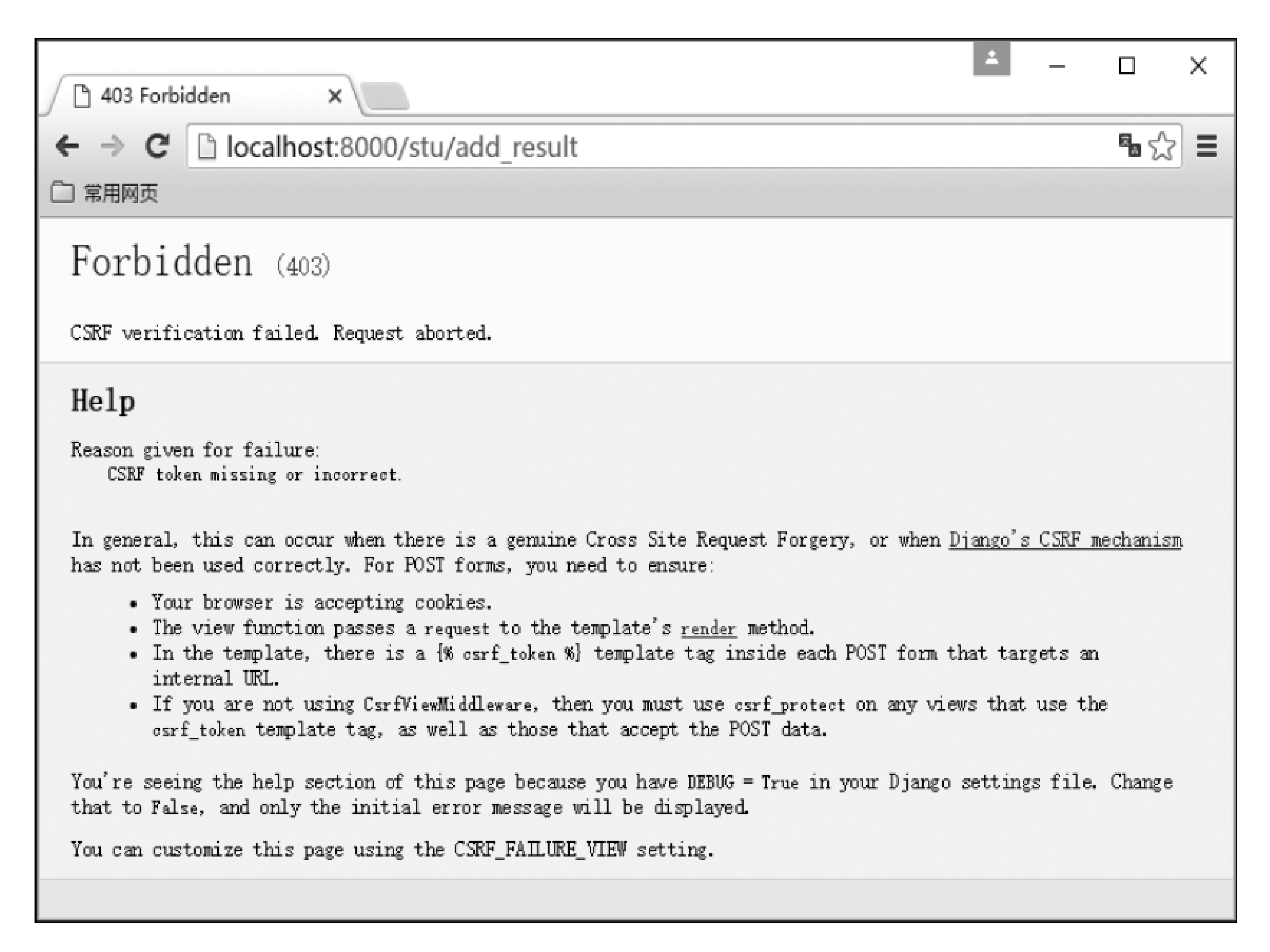


图 11-11 防 CSRF 请求错误演示

理方法。下面我们创建一个基本的数据模型,用于记录学生的基本信息。

在 stu 应用的 models. py 文件中编写如下代码:

```
from django.db import models


#Create your models here.
class Student(models.Model):
    stu_no=models.CharField(max_length=20)              #学号
    name=models.CharField(max_length=20)                #姓名
    sex=models.CharField(max_length=2)                  #性别
    major=models.CharField(max_length=50)               #专业
```

该段代码定义了一个名称为 Student 的类,该类继承自 models 中的 Model 类,在 Student 的类体中定义了 4 个字段分别用来描述学生的学号、姓名、性别和专业,使用 models 中的 CharFiled 函数来生成字段,并传入各字段的最大长度。

Django 中的 Model 将开发人员从繁琐的数据库操作中解放出来,在 Model 里开发者无须关注 SQL 语法,也不需要了解各种数据库里复杂的数据格式,只需要通过简单几行代码就可以实现和数据库的所有交互。

11.5.4 数据迁移

Django(1.7 以后的版本)有功能强大的数据迁移工具 migrate,migrate 可以记录对 Model 的每一次变更,并可以轻松回退到以前的 Model,这一点类似于代码管理工具 Git、SVN(感兴趣的读者可以查看 Django 的官方文档),在此基础上,我们可以在 Model 里动态地添加和删除数据表的字段,这对在项目开发中变更产品需求具有很大的帮助。

在我们对 Model 文件做了更新后，可以先运行 makemigrations 提交最近更新后的 Model，Django 会在应用的 migrations 目录下生成本次的迁移文件，查看并确定该迁移文件无误，再运行 migrate，可以将数据库更新到我们最新的 Model 状态。运行如下命令：

```
manage.py makemigrations
```

针对上一节设计的数据模型，命令行窗口会显示信息如图 11-12 所示。

```
C:\WINDOWS\system32\cmd.exe
C:\Python27\Scripts\xmustu>manage.py makemigrations
Migrations for 'stu':
  0001_initial.py:
    - Create model Student

C:\Python27\Scripts\xmustu>
```

图 11-12　用 makemigrations 命令检查 Model 是否有更新

此次的数据迁移只需要创建一个名为 Student 的表，确定无误后，输入如下命令：

```
manage.py migrate
```

运行上述命令后将看到如图 11-13 所示的信息，表示数据成功迁移到数据库中。

```
C:\WINDOWS\system32\cmd.exe
    - Create model Student

C:\Python27\Scripts\xmustu>manage.py migrate
Operations to perform:
  Apply all migrations: admin, stu, contenttypes, auth, sessions
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying sessions.0001_initial... OK
  Applying stu.0001_initial... OK

C:\Python27\Scripts\xmustu>
```

图 11-13　使用 migrate 命令成功迁移数据到数据库中

在安装了 SQLiteManager 软件后，通过该软件打开 xmustu 目录下的 db. sqlite3 文件，我们会发现 Django 已经自动为我们创建了图 11-14 所示的数据表，其中前面 10 张表是 Django 默认创建的，最后一张 stu_student 是根据我们自定义创建的，表名的命名规则是采用 app_model 的形式。

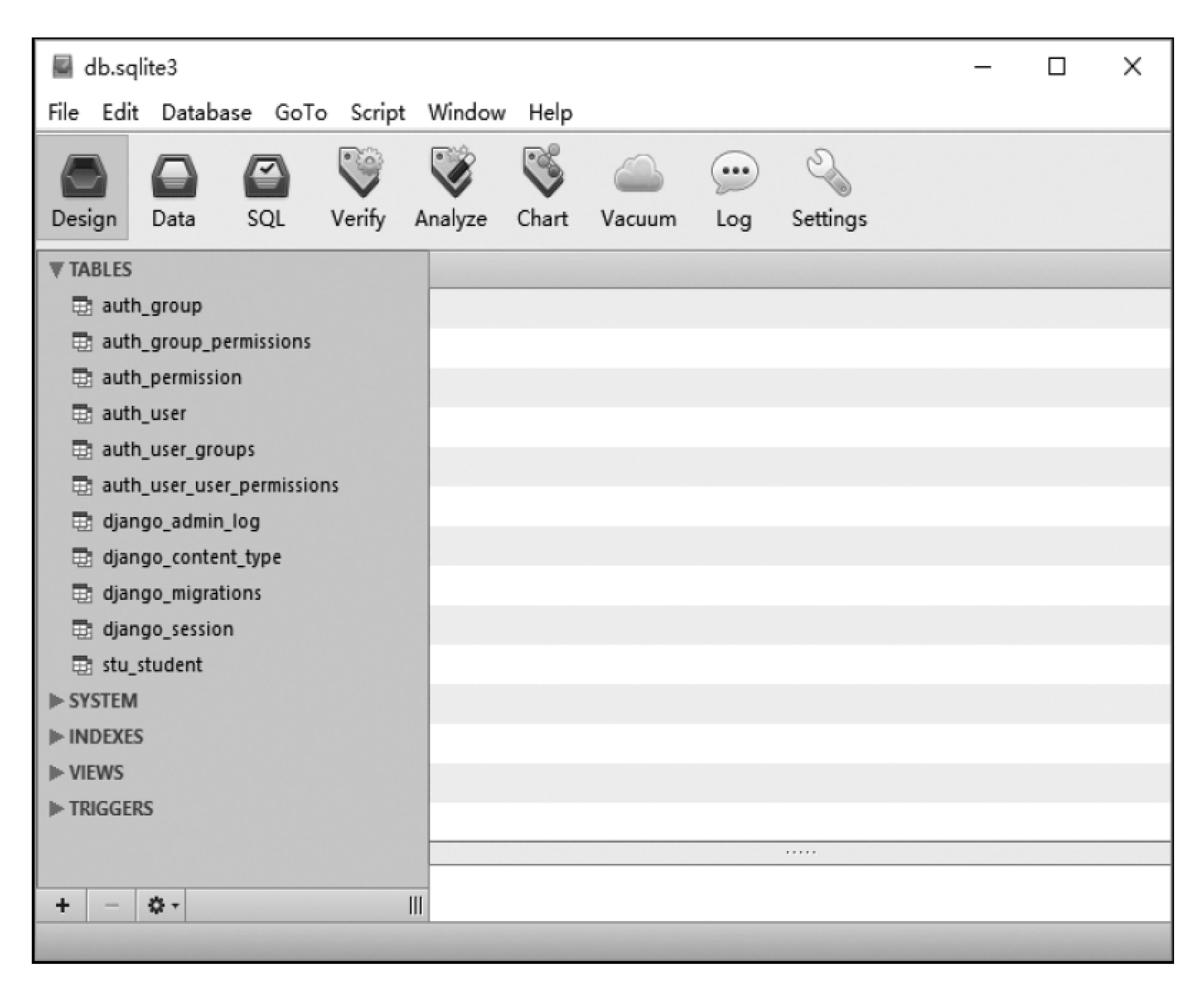


图 11-14　使用 SQLiteManager 查看数据库

11.6　Template 模板

11.6.1　什么是模板

Django 模板是一个 string 文本，它可以有效地分离一个文档的显示和数据，模板使用{{ varibles }}和表示多种逻辑的{% tags %}来规定文档如何显示，通常，模板用来输出 HTML 文本，但是 Django 模板也能生成其他基于文本的格式。Django 模板可以重用，从而减少代码的冗余和系统设计的复杂性。下面是一个简单的模板文件：

```
<html lang="zh_cn">
<head>
<title>{{ page_title }}</title>
</head>
<body>
<h3>学生列表</h3>
<div class="col-md-12">
<div class="table-responsive">
        <table class="table table-condensed">
```




```
            <thead>
              <tr>
                <th>#</th>
                <th>学号</th>
                <th>姓名</th>
                <th>性别</th>
                <th>专业</th>
              </tr>
            </thead>
            <tbody>
             {% for stu in stus %}
              <tr>
                <td>{{ forloop.counter }}</td>
                <td>{{ stu.stu_no }}</td>
                <td>{{ stu.name }}</td>
                <td>{{ stu.sex }}</td>
                <td>{{ stu.major }}</td>
              </tr>
              {% endfor %}
              </tbody>
          </table>
      </div>
</div>
</body>
</html>
```

这个模板本质上是 HTML,但是夹杂了一些变量和模板标签:

(1) 用{{ }}包围的是变量,如{{ page_title }},此处将会被给定变量的值替换,如何指定这些变量的值我们将在后面说明。

(2) 用{% %}包围的是逻辑块标签,如{% for stu in stus %} {% endfor %},标识一个 for 的循环区域块。

在 Python 代码中使用模板系统,请按照下面的步骤:

(1) 用模板代码创建一个 Template 对象,Django 也提供指定模板文件路径的方式创建 Template 对象。

(2) 使用一些给定变量 context 调用 Template 对象的 render()方法,这将返回一个完全渲染的模板,它是一个 string,其中所有的变量和块标签都会根据 context 得到值。

11.6.2　模板的继承

我们可以根据需要展示页面的排版。将页面划分为多个区域,并可以对每个区域设定一个专属模板,在渲染页面时,通过将所有的相关模板 include 进来。

如我们设定了一个 nav.html 用于表示导航栏,可以在最终的渲染模板中{% include 'nav.html' %}引入导航模板,诚然,include 可以有效减少模板的重复代码。但 Django 中

有一种更方便、更优雅的方式是模板继承(Template Inheritance)。

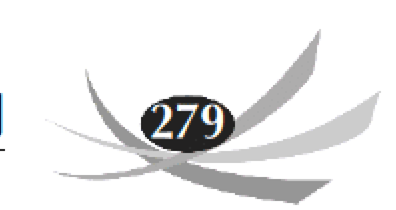
首先,在 stu 目录下创建一个 templates 文件夹,然后在该文件夹下创建父模板 base.html:

```
<html lang="zh_cn">
<head>
<title>{%block title %}{%endblock %}</title>
</head>
<body>
{%block content %}{%endblock %}
</body>
</html>
```

我们使用一个新的 tag,{% block %}用来告知模板引擎,这个部分会在子模板中实现。如果子模板没有实现,就会默认使用父模板的代码。

其次,在 templates 文件夹下创建子模板 home.html:

```
{%extends "base.html" %}
{%block title %} Template Inheritance {%endblock %}
{%block content %}
<div class="col-md-12">
<div class="table-responsive">
    <table class="table table-condensed">
      <thead>
        <tr>
          <th>#</th>
          <th>学号</th>
          <th>姓名</th>
          <th>性别</th>
          <th>专业</th>
        </tr>
      </thead>
      <tbody>
        {%for stu in stus %}
        <tr>
          <td>{{ forloop.counter }}</td>
          <td>{{ stu.stu_no }}</td>
          <td>{{ stu.name }}</td>
          <td>{{ stu.sex }}</td>
          <td>{{ stu.major }}</td>
        </tr>
         {%endfor %}
      </tbody>
</table>
</div>
```




```
</div>
{% endblock %}
```

只需要先使用标签{% extends %}继承父模板，再把相应需要实现的部分写上所需要的内容。最终该子模板继承父模板后渲染的结果的 body 标签的内容会和上一节中 HTML 代码的 body 标签的内容一样。

11.6.3　静态文件服务

Django 还提供静态文件服务，在 setting.py 的 INSTALLED_APPS 中有定义是否开启，如图 11-9 所示。部署项目时，我们常用的只读文件如 CSS 样式脚本、JavaScript 脚本和图片等文件可能根据每一个应用的独立性，而放在不同的子目录下，通过静态文件服务，我们可以不用关注具体的文件位置，仅仅通过 static 关键字就可以使服务器在 static 设定的所有目录下进行搜寻。

为了让本项目的排版显示更加美观，我们需要在 stu 应用中引入开源的自适应框架 Bootstrap，使其可以在移动设备上兼容显示。

使用自适应框架 Bootstrap，首先需要将其下载下来，这里下载的是 bootstrap-3.3.5 的压缩包。此外，Bootstrap 框架需要用到 JavaScript 的 jQuery 框架，这里下载的 jquery-2.2.0.min.js，同时也为了成功使用静态文件服务功能，需要定义静态文件的存放目录，这里的静态文件目录定义为 stu 应用目录下的 static 文件夹，将 bootstrap-3.3.5 的压缩包解压到该目录下，并把 jquery-2.2.0.min.js 文件放到 static 文件夹下的 jquery-2.2.0 目录。然后在 setting.py 文件添加如下代码：

```
STATIC_URL='/static/'
STATIC_ROOT=''
```

修改父模板 base.html，修改成如下所示：

```
{% load static %}
{% load compress %}
<!doctype html>
<html class="no-js" lang="zh_cn">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>{%block title %}{%endblock %}</title>
    <meta name="description" content="{%block description %}{%endblock %}">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    {% compress css %}
    <link rel="stylesheet" href="{% static 'bootstrap-3.3.5/css/bootstrap.min.css' %}"
>
    <link rel="stylesheet" href="{% static 'bootstrap-3.3.5/css/bootstrap-theme.min.
css' %}">
```

```
    {% endcompress %}

<!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries -->
    <!-- [if lt IE 9]>
      <script src="//cdn.bootcss.com/html5shiv/3.7.2/html5shiv.min.js"></script>
      <script src="//cdn.bootcss.com/respond.js/1.4.2/respond.min.js"></script>
    <![endif]-->
</head>
<body>
<!-- [if lt IE 8]>
<p class="browserupgrade">You are using an<strong>outdated</strong>browser. Please<a
href="http://browsehappy.com/">upgrade your browser</a>to improve your experience.</p>

<![endif]-->
<div>
<nav class="navbar navbar-inverse navbar-fixed-top" style="position:relative" role="
navigation">
    <div class="container">
        <div class="navbar-header">
                <button type="button" class="navbar-toggle collapsed" data-toggle="
                 collapse" data-target="#bs-example-navbar-collapse-1" aria-
                expanded="false">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
            <a class="navbar-brand" href="#">学生信息管理系统</a>
        </div>
        {% block menu %}{% endblock %}
        <!--/.navbar-collapse -->
    </div>
</nav>
{% block content %}{% endblock %}
</div>
{% compress js %}
<script src="{% static 'jquery-2.2.0/jquery-2.2.0.min.js' %}"></script>
<script src="{% static 'bootstrap-3.3.5/js/bootstrap.min.js' %}"></script>
{% endcompress %}
</body>
</html>
```

第一行代码的{% load static %}表示我们将加载服务器的静态文件服务。第二行的{% load compress %}表示我们将启用压缩，compress 压缩一般用于 CSS 和 JS，凡是使

用 compress 命令的区域，将会被压缩合并。如下代码，将会自动把两个 CSS 文件的内容合并生成一个新的 CSS 文件，新生成的 CSS 文件位于 static 对应的文件夹下：

```
{% compress css %}
<link rel="stylesheet" href="{% static 'libs/bootstrap-3.3.5/css/bootstrap.min.css' %}"
>
<link rel="stylesheet" href="{% static 'libs/bootstrap-3.3.5/css/bootstrap-theme.min.
css' %}">
{% endcompress %}
```

如下代码将自动把两个 JS 文件的内容合并生成为一个新的 JS 文件，新生成的 JS 文件位于 static 对应的文件夹下：

```
{% compress js %}
<script src="{% static 'libs/jquery/2.1.0/jquery-2.1.0.min.js' %}"></script>
<script src="{% static 'libs/bootstrap-3.3.5/js/bootstrap.min.js' %}"></script>
{% endcompress %}
```

使用 compress 压缩功能需要先安装 django_compressor，这里使用 pip 命令安装：

```
pip install django_compressor
```

注意：安装前要确保已安装 Microsoft Visual C++ Compiler for Python 2.7，如果没安装需要先将其下载，下载地址：https://www.microsoft.com/en-us/download/confirmation.aspx?id=44266，然后安装。

在 setting.py 文件的 INSTALLED_APPS 中添加 compressor 应用。

11.7　学生信息管理

11.7.1　查询学生

1. 创建视图

编辑视图文件 views.py，添加一个视图用于初始化学生列表，代码如下：

```
#新增加三条导入语句
from django.views.generic.base import TemplateView
from stu.models import Student
from django.core.paginator import Paginator,PageNotAnInteger,EmptyPage

class HomeView(TemplateView):                           #继承 TemplateView 类
    template_name='home.html'
    def get(self, request, *args, **kwargs):
        limit=20                                        #每页显示 20 条记录
        stus=Student.objects.all()
        paginator=Paginator(stus, limit)                #实例化一个分页对象
        page=request.GET.get('page')                    #获取页码
```

```
        try:
            stus=paginator.page(page)                    #获取某页对应的记录
        except PageNotAnInteger:                         #如果页码不是个整数
            stus=paginator.page(1)                       #取第一页的记录
        except EmptyPage:                                  #如果页码太大,没有相应的
记录
            stus=paginator.page(paginator.num_pages)   #取最后一页的记录
        context={
            'stus': stus,
        }
        return self.render_to_response(context)
```

其中,context 为 Json 格式的满足请求的学生信息。

2. 创建模板文件

修改前面创建的子模板 home.html,用于显示查询的学生列表,代码如下:

```
{%extends "base.html" %}
{%block title %}学生信息管理系统{%endblock %}
{%block content %}
<div  class="container">
<h3>学生列表</h3>
<div class="table-responsive">
  <table class="table table-condensed">
    <thead>
      <tr>
        <th>#</th>
        <th>学号</th>
        <th>姓名</th>
        <th>性别</th>
        <th>专业</th>
        <th>操作</th>
        </tr>
    </thead>
    <tbody>
        {%for stu in stus %}
        <tr>
            <td>{{ forloop.counter }}</td>
            <td>{{ stu.stu_no }}</td>
            <td>{{ stu.name }}</td>
            <td>{{ stu.sex }}</td>
            <td>{{ stu.major }}</td>
            <td><a href="/stu/edit?stu_no={{ stu.stu_no }}"><span class=
              "glyphicon glyphicon-edit glyphicon-btn" title="修改"></span></a>
                   <a href="/stu/del?stu_no={{ stu.stu_no }}"><span class=
                     "glyphicon glyphicon-trash glyphicon-btn" title="删除">
                     </span></a>
```

```
                </td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
</div>
</div>
{% endblock %}
{% block menu %}
<ul class="nav navbar-nav">
    <li class="active"><a href="/">学生列表</a></li>
    <li><a href="/stu/add">添加学生</a></li>
</ul>
{% endblock %}
```

在 urls.py 文件中，我们需要设定访问学生列表的 URL 链接，这里把它设定为系统的首页，修改后的 urls.py 文件代码为：

```
from django.conf.urls import url
from django.contrib import admin
from stu.views import HomeView

urlpatterns=[
    url(r'^admin/', admin.site.urls),
    url(r'^$ ', HomeView.as_view(), name='home'),
]
```

由于数据库的学生信息暂时为空，所以最终的渲染查询结果如图 11-15 所示。

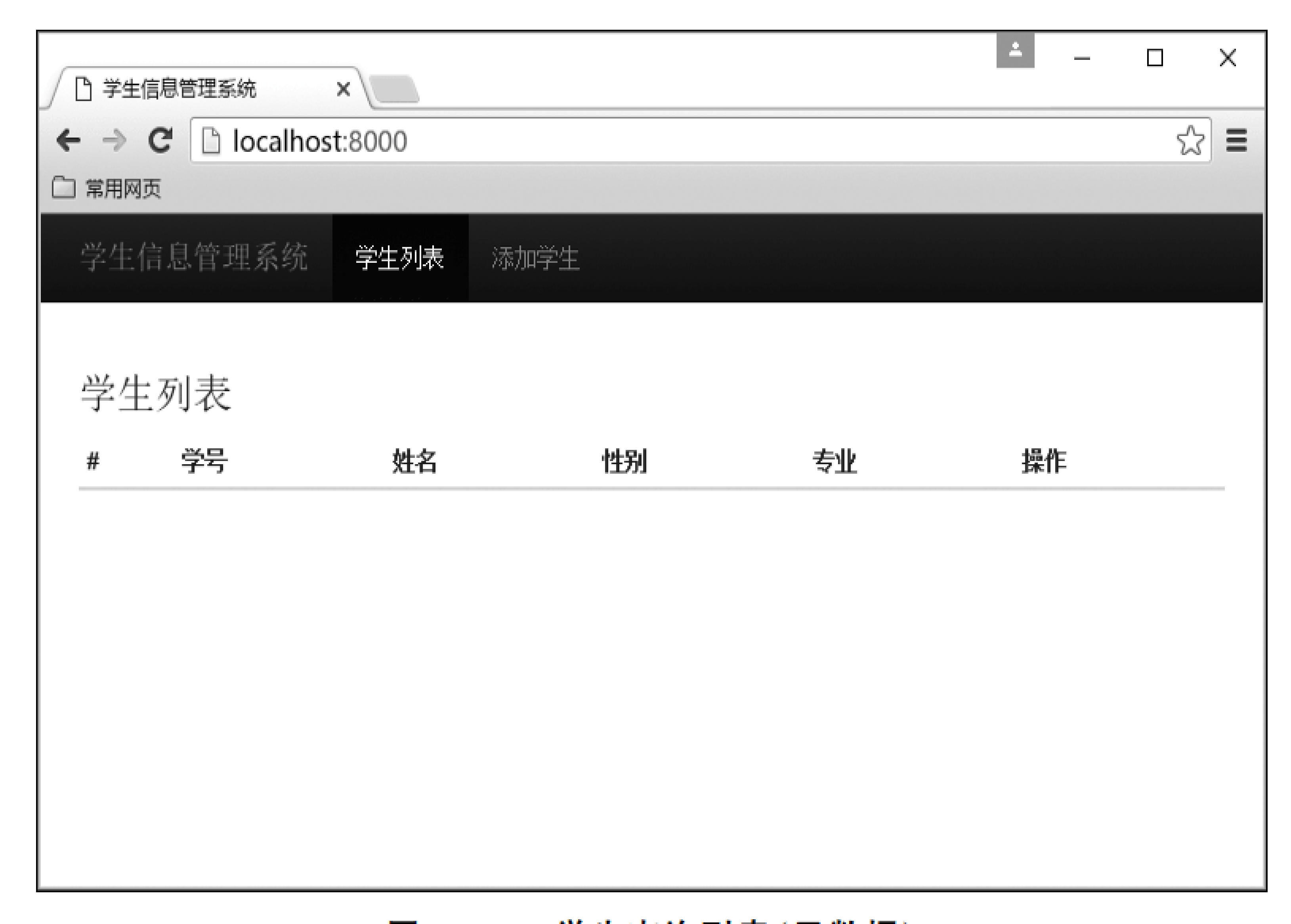


图 11-15　学生查询列表(无数据)

11.7.2 添加学生

在上一节中我们建立了一个学生信息查询分页显示列表，但是由于数据表暂时为空，所以我们看不到任何内容。这一节中，我们将实现在页面上向数据库添加数据的功能，我们需要在前端的页面中通过表单的输入来接收新学生。Django 已经构建好了通用 forms 包，我们可以借用 forms 包来构建和 Model 一致的学生表单，在 stu 目录下创建 form.py：

```
#coding:utf-8
from django import forms

class StuForm(forms.Form):
    stu_no=forms.CharField(
        label='学号', max_length=20,
        widget=forms.TextInput(attrs={'class': 'form-control', 'placeholder': '输入学号', '
        required': 'required',}),
        )                                                  #输入学号的文本框
    name=forms.CharField(
        label='姓名', max_length=20,
        widget=forms.TextInput(attrs={'class': 'form-control', 'placeholder': '输入姓名', '
        required': 'required',}),
        required=True)                                     #输入学号的文本框

    sex=forms.ChoiceField(
        choices=(('男', '男'),('女', '女'),),
        widget=forms.Select(attrs={'class': 'form-control'}),
        )                                                  #性别选择框
    major=forms.CharField(
        label='专业', max_length=100,
        widget=forms.TextInput(attrs={'class': 'form-control', 'placeholder': '输入专业', '
        required': 'required',}),
        required=True)                                     #输入学号的文本框
```

编辑视图文件 views.py，添加一个视图 stu_add 初始化添加学生信息的表单，代码如下：

```
#新增导入语句
from stu.form import StuForm
from django.template import loader,RequestContext
from django.http import HttpResponse

def stu_add(request):
    template=loader.get_template('stu_add.html')           #指定要渲染的模板
    form=StuForm()                                         #实例化一个学生表单
```

```
    context=RequestContext(request, {'form': form})
    return HttpResponse(template.render(context))
```

在 templates 目录下创建一个子模板 stu_add.html 用于填写学生信息的表单，注意在表单渲染显示中，我们添加了{% csrf_token %}，否则表单将会因为 Django 默认开启了防止跨站请求伪造而提交失败，如图 11-11 所示。

```
{% extends "base.html" %}
{% block title %}学生信息管理系统{% endblock %}
{% block content %}
<div  class="container">
<h3>添加新学生</h3>
<form role="form" action="{% url 'stu_add_result' %}" method="POST">
<div class="form-group">
    <label for="Stu_No">学号</label>
    {{ form.stu_no }}
  </div>
  <div class="form-group">
    <label for="Stu_Name">姓名</label>
     {{ form.name }}
  </div>
  <div class="form-group">
      <label for="Stu_Sex">性别</label>
      {{ form.sex }}
  </div>
  <div class="form-group">
      <label for="Stu_Major">专业</label>
       {{ form.major }}
    </div>
    {% csrf_token %}
    <button type="submit" class="btn btn-primary">提交</button>
  </form>
<div>
{% endblock %}
{% block menu %}
<ul class="nav navbar-nav">
      <li><a href="/">学生列表</a></li>
      <li  class="active"><a href="/stu/add">添加学生</a></li>
</ul>
{% endblock %}
```

在 urls.py 中，我们需要设定访问新增学生页面的 URL 链接，代码如下：

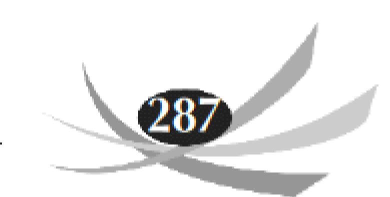
```
from stu import views                                        #新增导入语句
url(r'^stu/add$ ', views.stu_add, name='stu_add'),          #新增 URL
```

此外，我们还需要一个页面处理提交给服务器的新增学生的表单信息。编辑视图文件 views.py，添加一个视图 stu_add_result 用于处理提交新增学生的表单信息，并显示处理结果，代码如下：

```
def stu_add_result(request):
    template=loader.get_template('stu_add_result.html')          #指定要渲染的模板
    if request.method=='POST':
        form=StuForm(request.POST)
        #check whether it's valid:
        if form.is_valid():
            stu_no=form.cleaned_data['stu_no']
            name=form.cleaned_data['name']
            sex=form.cleaned_data['sex']
            major=form.cleaned_data['major']
            try:
                stu=Student.objects.get(stu_no=stu_no)     #检查是否学号重复了
                message='已经存在该学号的学生'
                alert_class='alert-warning'          #Bootstrap中用于显示警告的样式类
            except Student.DoesNotExist:
                stu=Student(
                    stu_no=stu_no,
                    name=name,
                    sex=sex,
                    major=major)
                stu.save()
                message='成功添加'
                alert_class='alert-success'          #Bootstrap中用于显示成功的样式类
            result={
                'alert_class': alert_class,
                'message': message,
                'stu_no': stu.stu_no,
                'name': stu.name,
                'sex': stu.sex,
                'major': stu.major,
            }
    context=RequestContext(request, result)
    return HttpResponse(template.render(context))
```

在 templates 目录下创建一个子模板 stu_add_result.html 用于显示添加新学生的结果，代码如下：

```
{%extends "base.html"%}
{%block title%}学生信息管理系统{%endblock%}
{%block content%}
<div class="container">
```




```
<h3>添加学生</h3>
<div class="alert {{alert_class}} alert-dismissible fade in" role="alert">
<h4><strong>{{ message }}</strong></h4>
<p>学号:{{ stu_no }}</p>
<p>姓名:{{ name }}</p>
<p>性别:{{ sex }}</p>
<p>专业:{{ major }}</p>
</div>
</div>
{% endblock %}
{% block menu %}
<ul class="nav navbar-nav">
    <li><a href="/">学生列表</a></li>
    <li  class="active"><a href="/stu/add">添加学生</a></li>
</ul>
{% endblock %}
```

在 urls.py 中,我们需要设定新增学生表单提交响应页面的 URL 链接,代码如下:

```
url(r'^stu/add_result$ ', views.stu_add_result, name='stu_add_result'),
#新增 URL
```

在浏览器中输入"http://localhost:8000/stu/add"或者在首页单击添加学生菜单按钮,可以看到最终添加学生页面的渲染结果如图 11-16 所示。



图 11-16　渲染添加新学生的表单页面

填写新生张三的数据并提交，如图 11-17 所示。

学生信息管理系统
localhost:8000/stu/add
常用网页
学生信息管理系统 学生列表 添加学生
添加新学生
学号
2016001
姓名
张三
性别
男
专业
计算机科学与技术
提交

图 11-17　添加学生张三的信息

提交处理结果如图 11-18 所示。



图 11-18　添加学生张三的处理结果

继续填写新生李四的数据并提交，如图 11-19 所示。

学生信息管理系统
localhost:8000/stu/add
常用网页
学生信息管理系统　学生列表　添加学生
添加新学生
学号
2016002
姓名
李四
性别
女
专业
经济学
提交

图 11-19　添加学生李四的信息

提交处理结果如图 11-20 所示。

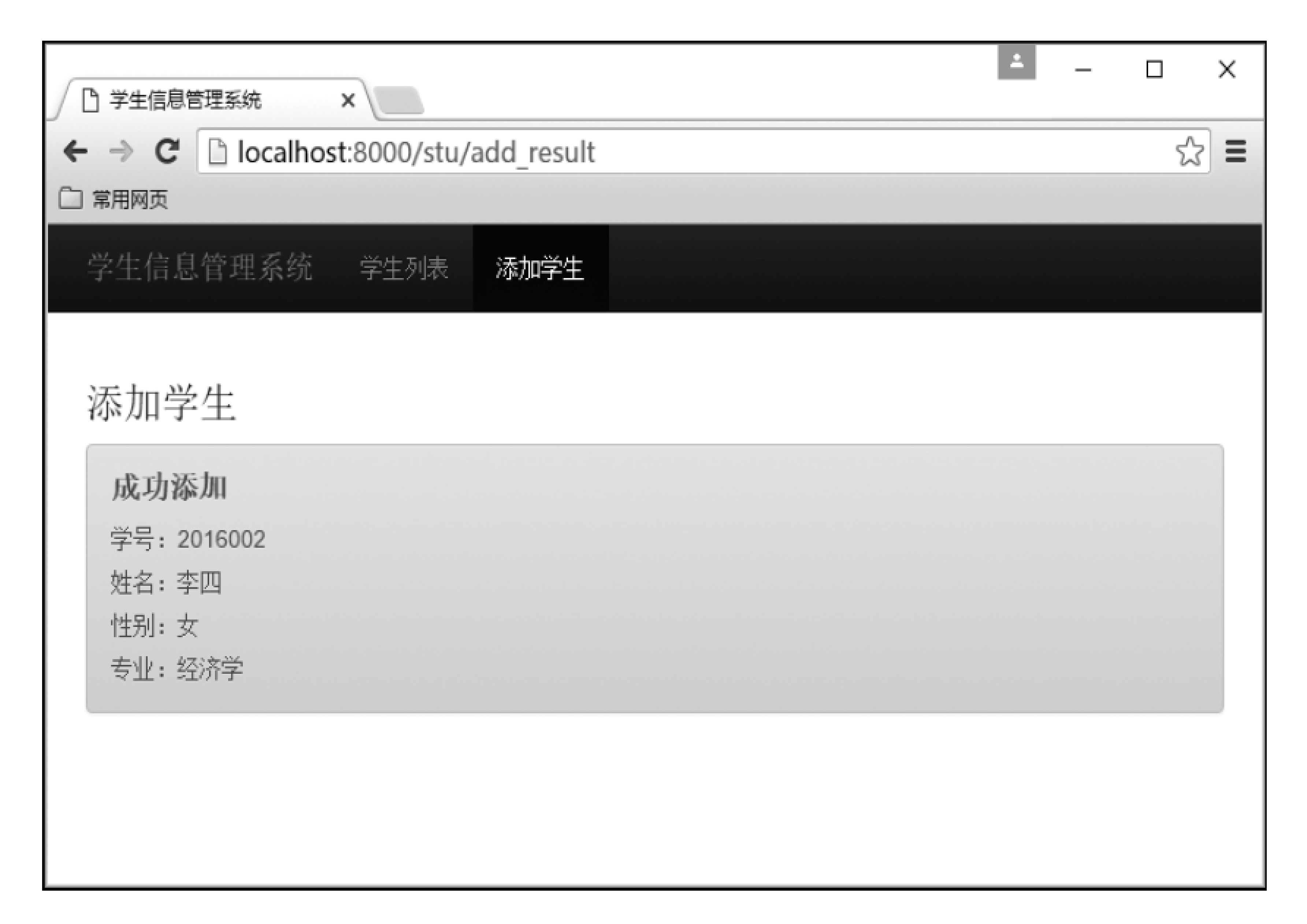


图 11-20　添加学生李四的处理结果

若重复提交，将会提示该学号学生已存在，如图 11-21 所示。

成功添加两名新生后，单击学生列表菜单按钮，返回首页，可以看到此时已有两条学生的记录，如图 11-22 所示。



图 11-21　重复添加学生李四的处理结果

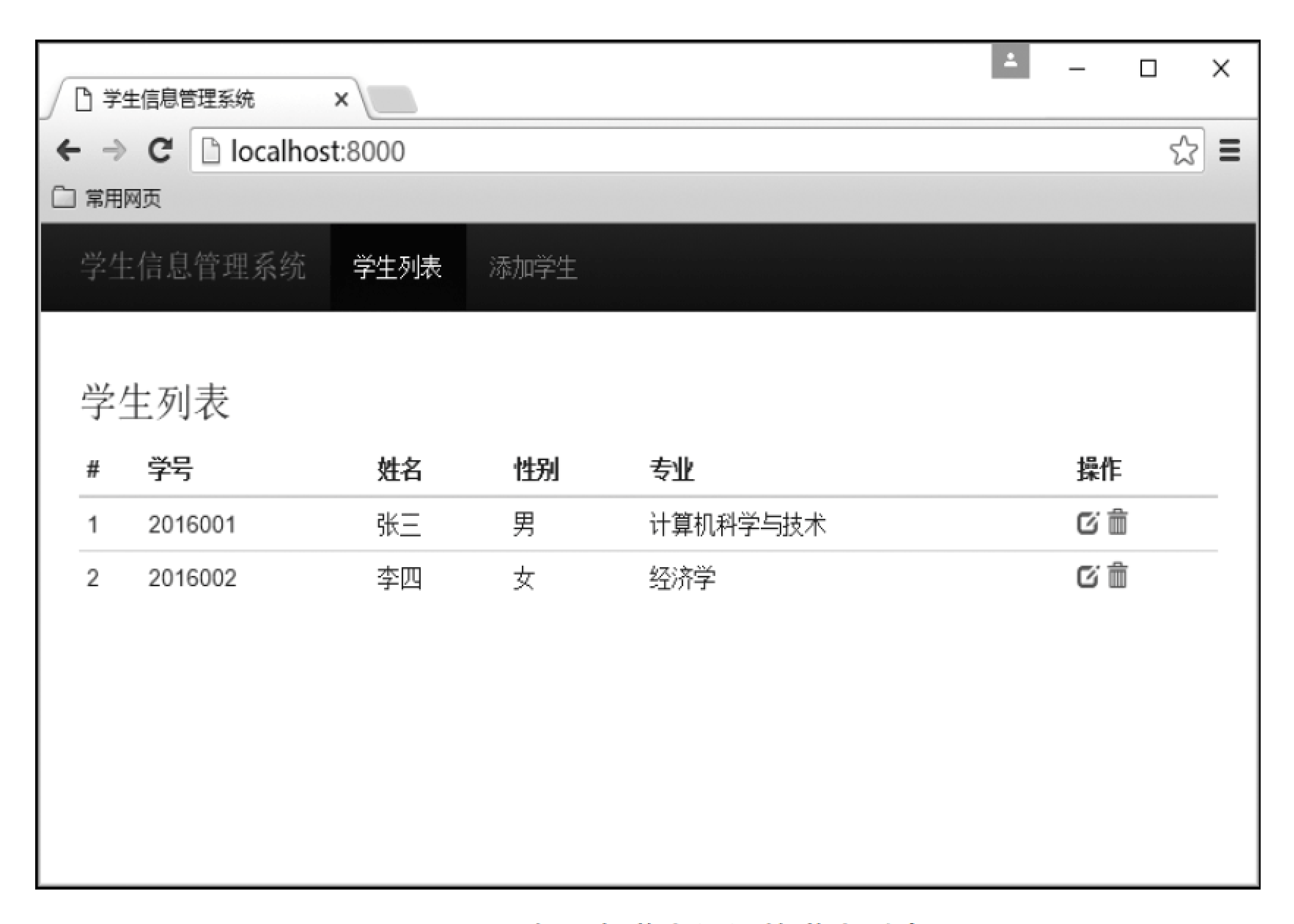
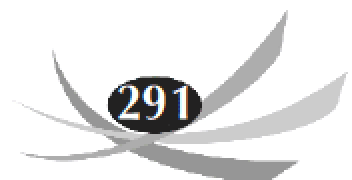


图 11-22　有两名学生记录的学生列表

11.7.3　修改学生

编辑视图文件 views.py，添加一个视图 stu_edit 用于处理初始化需要编辑的学生信息，代码如下：

```
def stu_edit(request):
    template=loader.get_template('stu_edit.html')
    form=StuForm()
```




```
    stu_no=request.GET.get('stu_no')                    #接收需要编辑的学生学号
    try:
        stu=Student.objects.get(stu_no=stu_no)
        form.fields["stu_no"].initial=stu.stu_no         #根据查询结果初始化表单中的
学号
        form.fields["name"].initial=stu.name             #根据查询结果初始化表单中的
姓名
        form.fields["sex"].initial=stu.sex               #根据查询结果初始化表单中的
性别
        form.fields["major"].initial=stu.major           #根据查询结果初始化表单中的
专业
        exist=True
        message='存在该学号'
        alert_class='alert-success'                  #Bootstrap中用于显示成功的样式类
    except Student.DoesNotExist:
        exist=False
        message='不存在该学号'
        alert_class='alert-warning'                  #Bootstrap中用于显示警告的样式类
    result={
        'alert_class': alert_class,
        'message': message,
        'stu_no': stu_no,
        'exist': exist,
        'form': form,
     }
    context=RequestContext(request, result)
    return HttpResponse(template.render(context))
```

在 templates 目录下创建一个子模板 stu_edit.html 用于填写需要修改学生的信息表单，代码如下：

```
{%extends "base.html" %}
{%block title %}学生信息管理系统{%endblock %}
{%block content %}
<div  class="container">
<h3>修改学生{{ stu_no }}</h3>
{%if exist %}
<form role="form" action="{%url 'stu_edit_result' %}" method="POST">
<div class="form-group">
    <label for="Stu_No">学号</label>
    {{ form.stu_no }}
</div>
<div class="form-group">
    <label for="Stu_Name">姓名</label>
    {{ form.name }}
```

```
    </div>
    <div class="form-group">
      <label for="Stu_Sex">性别</label>
      {{ form.sex }}
    </div>
    <div class="form-group">
      <label for="Stu_Major">专业</label>
      {{ form.major }}
    </div>
      {% csrf_token %}
    <button type="submit" class="btn btn-primary">提交</button>
  </form>
  {% else %}
  <div class="alert {{alert_class}} alert-dismissible fade in" role="alert">
    {{ message }}
  </div>
  {% endif %}
  {% endblock %}
  {% block menu %}
  <ul class="nav navbar-nav">
      <li><a href="/">学生列表</a></li>
      <li  class="active"><a href="#">修改学生</a></li>
  </ul>
  {% endblock %}
```

在 urls.py 中，我们需要设定访问修改学生页面的 URL 链接，代码如下：

```
url(r'^stu/edit$ ', views.stu_edit, name='stu_edit'),        #新增 url
```

此外，我们还需要一个页面处理提交给服务器的修改学生的表单信息。编辑视图文件 views.py，添加一个视图 stu_edit_result 用于处理提交新增学生的表单信息，并显示处理结果，代码如下：

```
def stu_edit_result(request):
    template=loader.get_template('stu_edit_result.html')     #指定要渲染的模板
    if request.method=='POST':
        form=StuForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            stu_no=form.cleaned_data['stu_no']                #获取学号
            name=form.cleaned_data['name']                    #获取姓名
            sex=form.cleaned_data['sex']                      #获取性别
            major=form.cleaned_data['major']                  #获取专业
            try:
                stu=Student.objects.get(stu_no=stu_no)        #按学号查询
                stu.name=name
```

```
            stu.sex=sex
            stu.major=major
            stu.save()                        #写入修改后的新记录
            message='成功修改'
            alert_class='alert-success'       #Bootstrap 中用于显示成功的样式类
        except Student.DoesNotExist:
            name=None
            sex=None
            major=None
            message='Does Not Exist '+stu_no
            alert_class='alert-warning'       #Bootstrap 中用于显示警告的样式类
        result={
            'alert_class': alert_class,
            'message': message,
            'stu_no': stu_no,
            'name': name,
            'sex': sex,
            'major': major,
        }
    context=RequestContext(request, result)
    return HttpResponse(template.render(context))
```

在 templates 目录下创建一个子模板 stu_edit_result.html 用于显示修改学生的结果，代码如下：

```
{%extends "base.html" %}
{%block title %}学生信息管理系统{%endblock %}
{%block content %}
<div class="container">
<h3>修改学生</h3>
<div class="alert {{alert_class}} alert-dismissible fade in" role="alert">
  <h4><strong>{{ message }}</strong></h4>
  <p>学号:{{ stu_no }}</p>
  <p>姓名:{{ name }}</p>
  <p>性别:{{ sex }}</p>
  <p>专业:{{ major }}</p>
</div>
</div>
{%endblock %}
{%block menu %}
<ul class="nav navbar-nav">
    <li><a href="/">学生列表</a></li>
    <li  class="active"><a href="#">修改学生</a></li>
</ul>
{%endblock %}
```

在 urls.py 中，我们需要设定提交修改学生信息表单的 URL 链接，代码如下：

```
url(r'^stu/edit_result$ ', views.stu_edit_result, name='stu_edit_result'),     #新增 URL
```

通过单击学生列表的修改按钮，触发修改学生操作，传递的 URL 链接中添加参数 stu_no=2016001，将修改学号为 2016001 的学生记录，如图 11-23 所示。

图 11-23　修改学生张三的信息

把张三的专业修改为“会计学”，提交后的处理结果如图 11-24 所示。

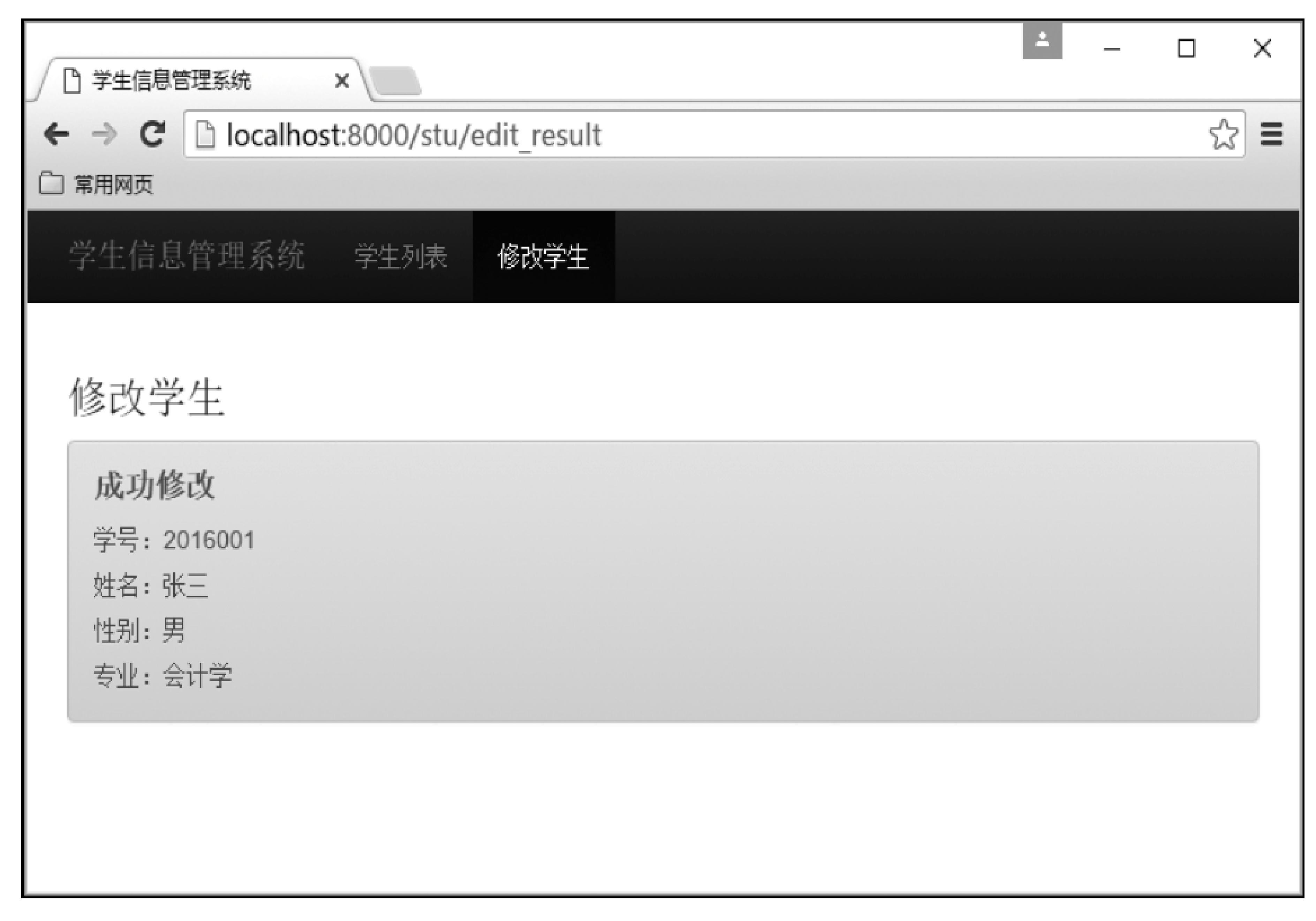
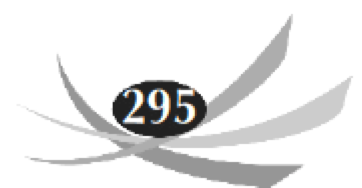


图 11-24　修改学生张三信息的处理结果



修改成功后，学生列表信息已是修改后的信息，如图 11-25 所示。

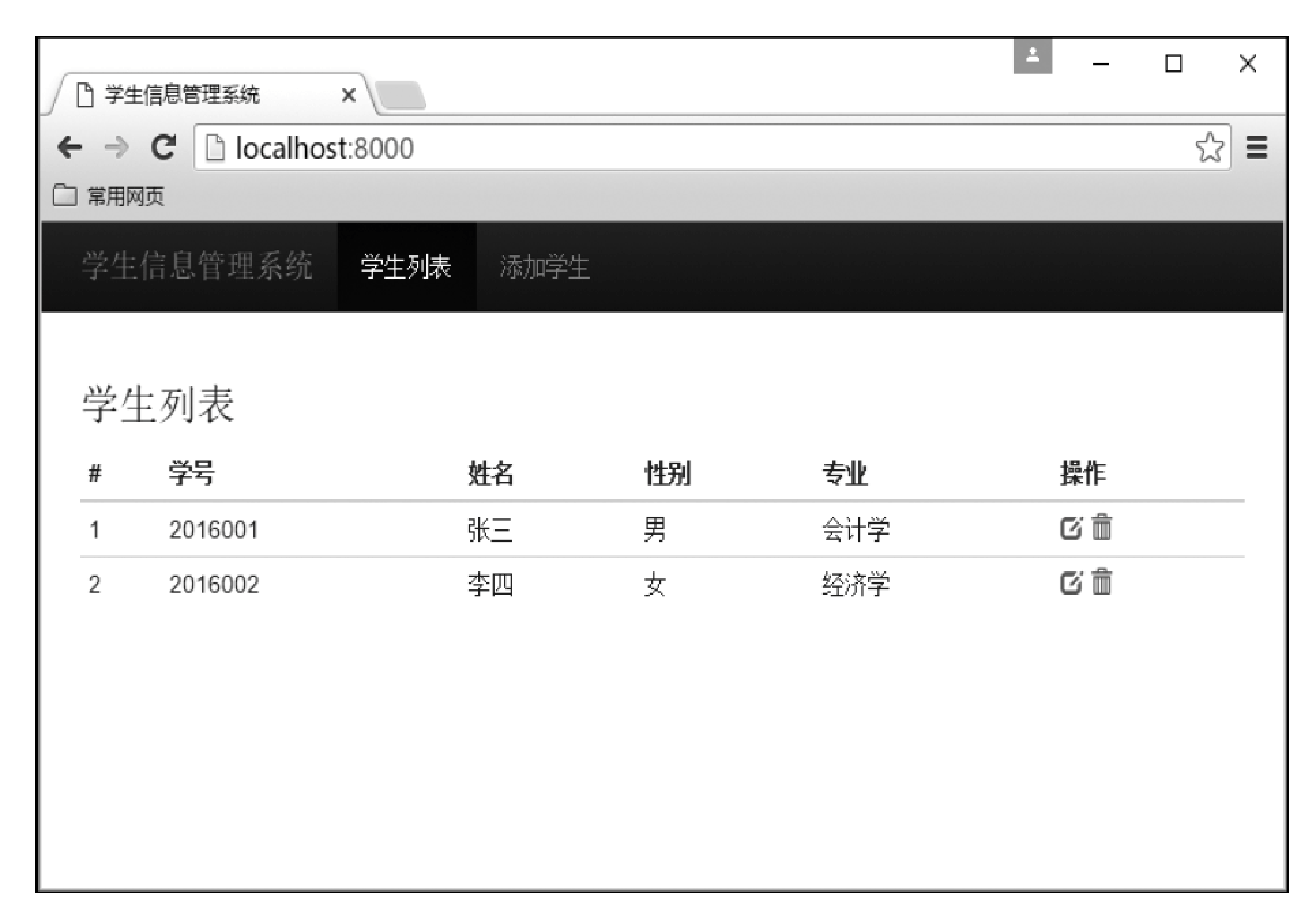


图 11-25　修改学生张三信息后的学生列表

注意：学生的学号不能修改，如果试图把张三的学号修改为 2016003，将提示不存在此学号的学生，如图 11-26 所示。

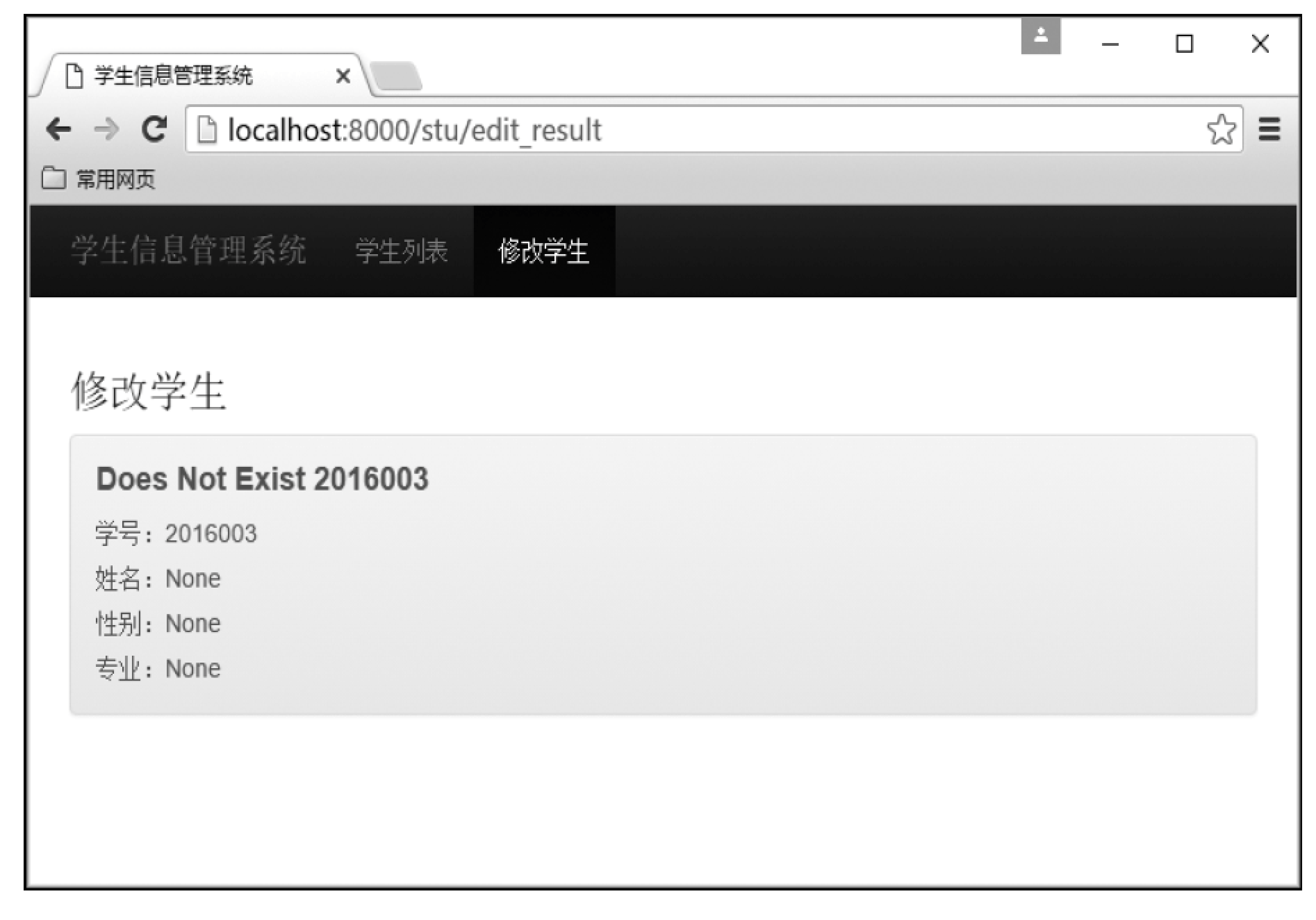


图 11-26　修改学生的学号的处理结果

11.7.4 删除学生

编辑视图文件 views.py，添加一个视图 stu_del 用于处理欲删除的学生，代码如下：

```
def stu_del(request):
    template=loader.get_template('stu_del.html')
    stu_no=request.GET.get('stu_no')
    try:
        stu=Student.objects.get(stu_no=stu_no)
        stu.delete()
        message='Delete %s Success' %stu_no
        alert_class='alert-success'
    except Student.DoesNotExist:
        message='Does Not Exist '+stu_no
        alert_class='alert-warning'
    result={
        'alert_class': alert_class,
        'message': message,
    }
    context=RequestContext(request, result)
    return HttpResponse(template.render(context))
```

在 templates 目录下创建一个子模板 stu_del.html 用于显示删除学生的结果，代码如下：

```
{%extends "base.html" %}
{%block title %}学生信息管理系统{%endblock %}
{%block content %}
<div class="container">
<h3>删除学生</h3>
<div class="alert {{alert_class}} alert-dismissible fade in" role="alert">
  <h4><strong>{{ message }}</strong></h4>
</div>
</div>
{%endblock %}
{%block menu %}
<ul class="nav navbar-nav">
    <li><a href="/">学生列表</a></li>
    <li  class="active"><a href="#">删除学生</a></li>
</ul>
{%endblock %}
```

在 urls.py 中，我们需要设定提交修改学生信息表单的 URL 链接，代码如下：

```
url(r'^stu/del$ ', views.stu_del, name='stu_del'),        #新增 URL
```

通过单击学生列表的删除按钮,触发删除学生操作,传递的 URL 链接中添加参数 stu_no=2016001,将删除学号为 2016001 的学生记录,如图 11-27 所示。

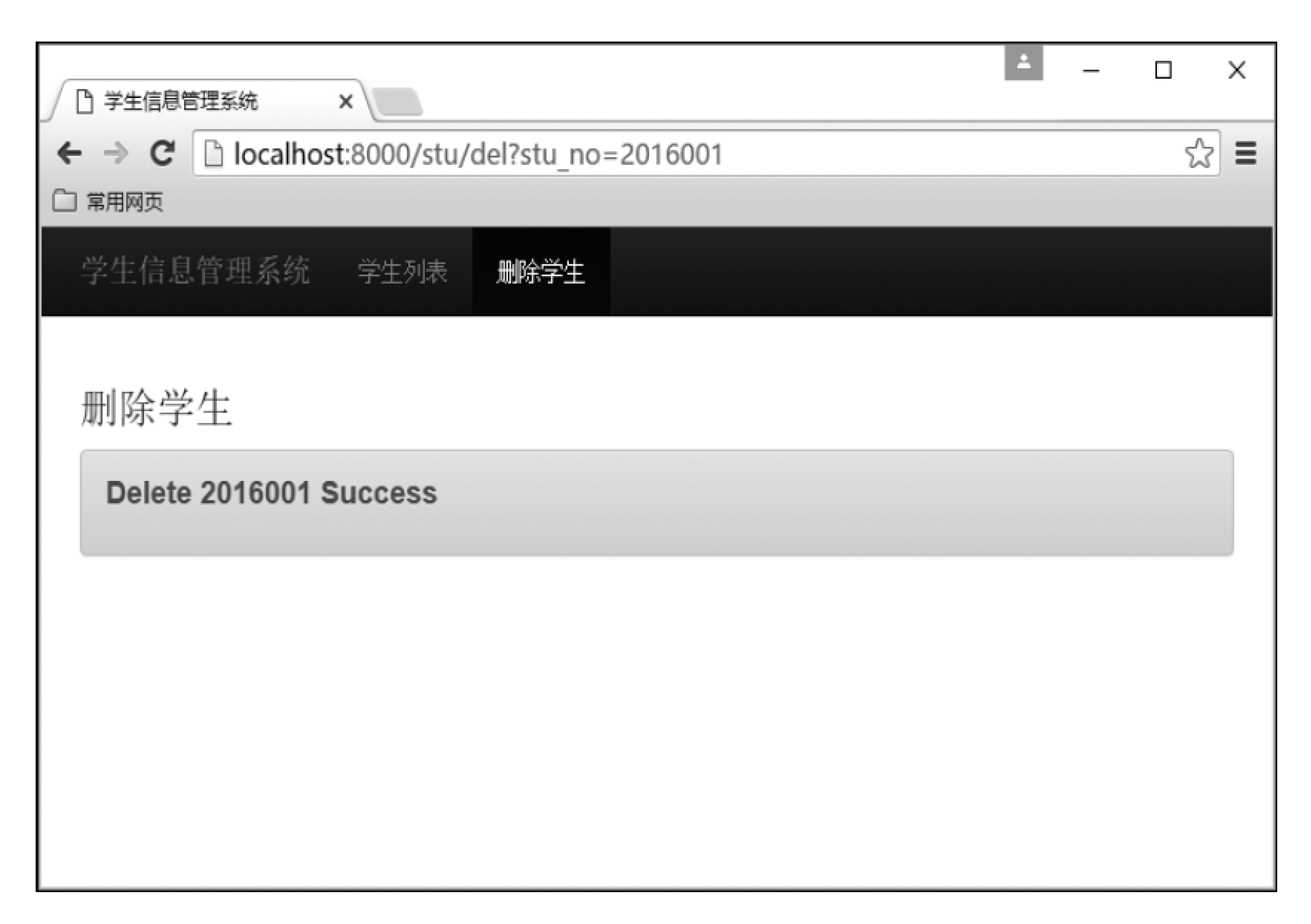


图 11-27　删除学生张三信息的处理结果

删除成功后,学生列表信息已是删除后的信息,如图 11-28 所示。

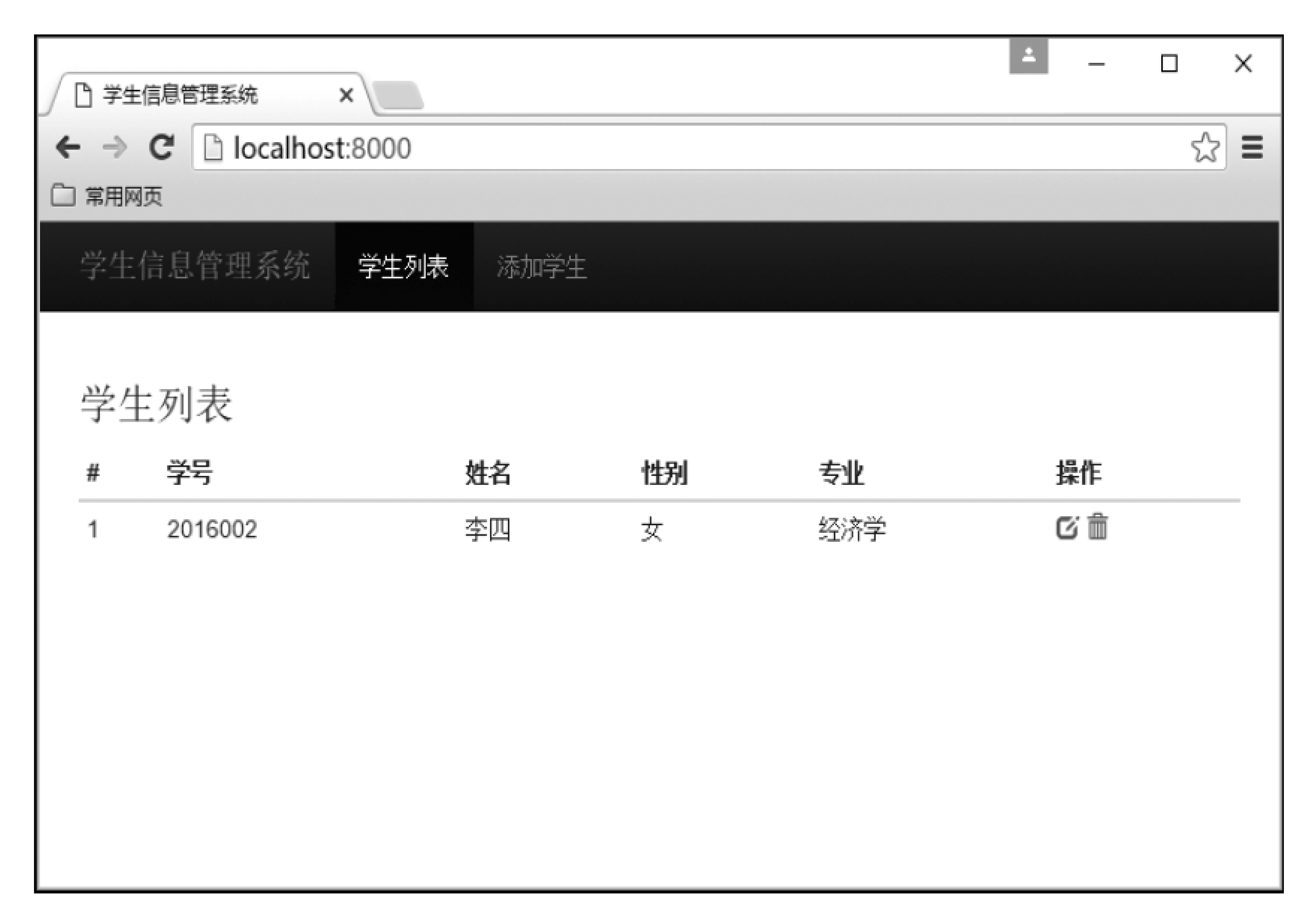


图 11-28　删除学生张三信息后的学生列表

当成功删除学生信息后再次刷新界面时会提示此学生不存在,如图 11-29 所示。

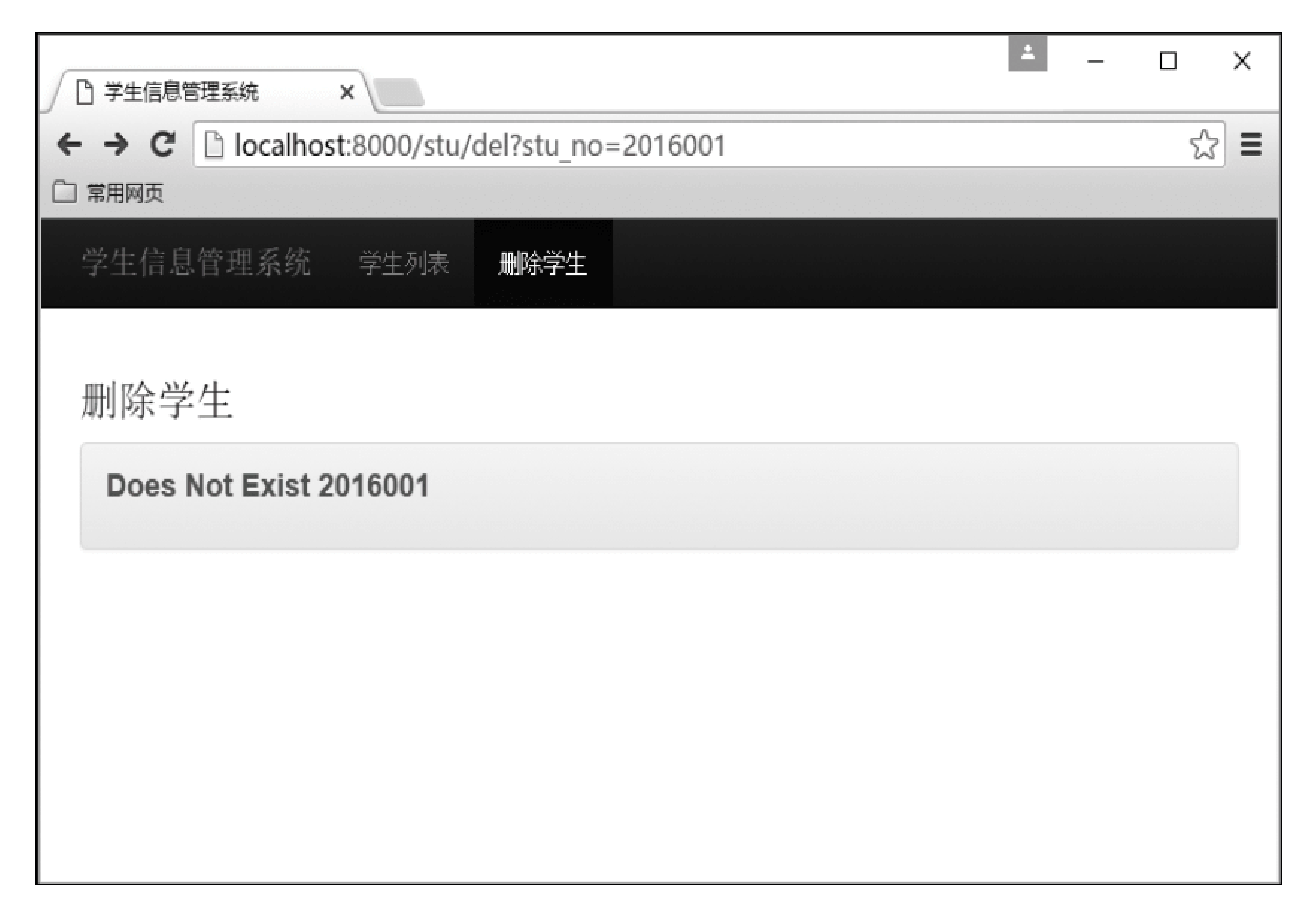


图 11-29 删除学生信息后试图再次删除的处理结果

11.8 本章小结

本章主要讲解了以下几个知识点。

(1) Django。Django是由Python开发的应用于Web开发的免费开源的高级动态语言框架。Django拥有完善的模板机制、对象关系映射机制以及拥有动态创建后台管理界面的功能。使用Django框架来开发Web应用,可以快速设计和开发具有MVC层次的Web应用。Django框架具有丰富的组件、对象关系映射和多数据库支持、简洁的URL设计、自动化的管理界面、强大的开发环境等特点。

(2) MVC模式。MVC是指现代程序设计中的一种分层设计模式,将传统程序按其功能边界分为表现层、逻辑层和控制层三部分,这种方式的划分能使程序设计变得更加容易,缩短了程序开发周期,同时开发出来的程序也易于维护。

(3) Django的MTV模式。Django的MTV模式是指由模型(Model)、模板(Template)和视图(View)三者有机组合形成的一种类似MVC的分层模式。其中,M代表模型,即数据存取层,该层处理与数据相关的所有事务;T代表模板,即表现层,该层处理与表现相关的逻辑;V代表视图,即业务逻辑层,该层包含存取模型及调取恰当模板的相关逻辑。当浏览器发出请求时,会根据URL地址匹配相应的视图,视图会调用相应的模型和模板,处理完业务逻辑后,把渲染后的界面返回给浏览器,对请求做出响应。

(4) Django安装。Django的安装可以通过setup.py install命令或者pip install Django==1.9.1命令安装,但后者更方便。

(5) 创建Django开发项目。创建Django开发项目需要用django-admin.py startproject projectname(项目名)命令创建。创建项目后会自动生成相应的文件,如用于配置的setting.py文件、用于设置URL的urls.py文件等。启动Django开发项目的服

务器需要使用 manage. py runserver 命令。

(6) 创建项目应用。创建项目应用需要使用 manage. py startapp appname(应用名)。创建项目应用后会自动生成相应的文件,如用于定义数据模型的 models. py 文件、用于定义视图的 views. py 文件等。

(7) 数据迁移。1.7 以后版本的 Django 有功能强大的数据迁移工具 migrate。在我们对 models. py 文件做了更新后,可以先运行 manage. py makemigrations 命令提交最近更新后的 Model,Django 会在应用的 migrations 目录下生成本次的迁移文件,查看并确定该迁移文件无误,再运行 manage. py migrate,可以将数据库更新到我们最新的 Model 状态。

(8) 模板。Django 模板是一个 string 文本,它可以有效地分离一个文档的显示和数据,模板使用{{ varibles }}和表示多种逻辑的{% tags %}来规定文档如何显示,通常,模板用来输出 HTML 文本,但是 Django 模板也能生成其他基于文本的格式。Django 模板可以重用,从而减少代码的冗余和系统设计的复杂性。模板可以继承,在父模板中用{% block %}告知模板引擎,这个部分会在子模板中实现。在子模板中用{% extends "xxx"%}表示继承 xxx 父模板,再把相应需要实现的部分写上所需要的内容。

11.9　习　　题

一、解答题

1. 什么是 Django? 它有哪些特点?
2. 什么是 MVC 模式?
3. 什么是 MTV 模式?
4. 分别使用什么命令创建 Django 开发项目和运行其服务器?

二、上机练习

安装 Django 框架,部署运行本章项目代码,实现学生信息的管理。